saltfppipe Documentation

Release 0.0.1

Carl J. Mitchell

September 19, 2016

Contents

Welcome to the documentation for *saltfppipe*, a data reduction pipeline for the SALT Fabry-Perot system.

Note: If you make use of this code, please include the following in your acknowledgements section: "This research made use of *saltfppipe*, a data reduction package for the SALT Fabry-Perot."

This code is built atop IRAF/Pyraf, PySALT, Astropy, matplotLib, and SciPy. You should also cite/acknowledge these packages as appropriate.

Note: This software was developed primarily to reduce FP data taken in the medium-resolution mode and most of the testing has been with that data. It has been used to reduce low-resolution data to some degree of success. This code is **not** intended to be used on duel-etalon high-resolution data.

CHAPTER 1

Contents:

1.1 Installation

1.1.1 Downloading the Package

The latest version of *saltfppipe* is always available from its page on github. Use the green "Clone or Download" button to either clone the git repository on your local system or download the project as a .zip archive.

1.1.2 Building the Optional *voigtfit* module

saltfppipe comes with an optional python module called *voigtfit*, which is built from compiled Fortran code. The module is designed to fit Voigt profiles to the H-alpha line of excited hydrogen and (optionally) the [NII] line 21 Angstroms redward of H-alpha.

If you are using *saltfppipe* to produce H-alpha velocity fields, you'll want to build this module.

If you are using *saltfppipe* only to produce data cubes, the module is unnecessary and you can skip ahead.

In the cloned respository or unzipped directory, execute the following commands:

```
$ cd voigtfit
$ make
$ cd ..
```

Note: *voigtfit* requires the *f2py* and *gfortran* packages to be installed

1.1.3 Installing the Package

To install the package, run the following command in the cloned repository or unzipped directory:

```
$ python setup.py install
```

This will install the *saltfppipe* package in your python packages library.

If you built the optional *voigtfit* module, it will be installed automatically.

Note: If your python installation is root-protected, you may need to execute the install command as sudo.

1.1.4 Dependencies

The following is a list of packages on which *saltfppipe* depends. Several of them are either default packages or are readily availa

- distutils
- setuptools
- sys
- os
- shutil
- cStringIO
- numpy
- scipy
- matplotlib
- astropy
- iraf
- pyraf
- pysalt - Note: Use the "nightly build", not v0.47!
- zsalt
- photutils

1.2 Using the Pipeline

1.2.1 Starting Up

The *saltfppipe* pipeline is designed to be run in the directory above a 'raw/' directory filled with images from the telescope.

You can call it from an interactive environment like python or ipython with:

```
>>> from saltfppipe import pipeline
>>> pipeline()
```

Or you can call it directly from the command line with:

```
$ python relative/path/to/saltfppipe/pipeline.py
```

You can also run the pipeline from another location in the directory structure (other than one directory above the 'raw/' image directory:

```
>>> from saltfppipe import pipeline
>>> pipeline('relative/path/to/raw/')
```

Or again from the command line with the path to the raw directory as a calling argument:

```
$ python relative/path/to/saltfppipe/pipeline.py path/to/raw/)
```

1.2.2 Creating the 'product/' directory

The first step of *saltfppipe* is to convert the raw images in 'raw/' to pre-processed images in a 'product/' directory.

If you already have a 'product/' directory, you'll be greeted with the prompt:

```
Product directory already exists. Recreate it? (y/n)
```

If you downloaded the 'product/' directory directly from the SALT servers, you should answer **YES** to this question. The *saltfppipe* and *zSALT* packages create the product directory in a different way than the default SALT pre-processing.

If you have already created a 'product/' directory using *saltfppipe* and don't want to start over, you can answer **NO** to this question.

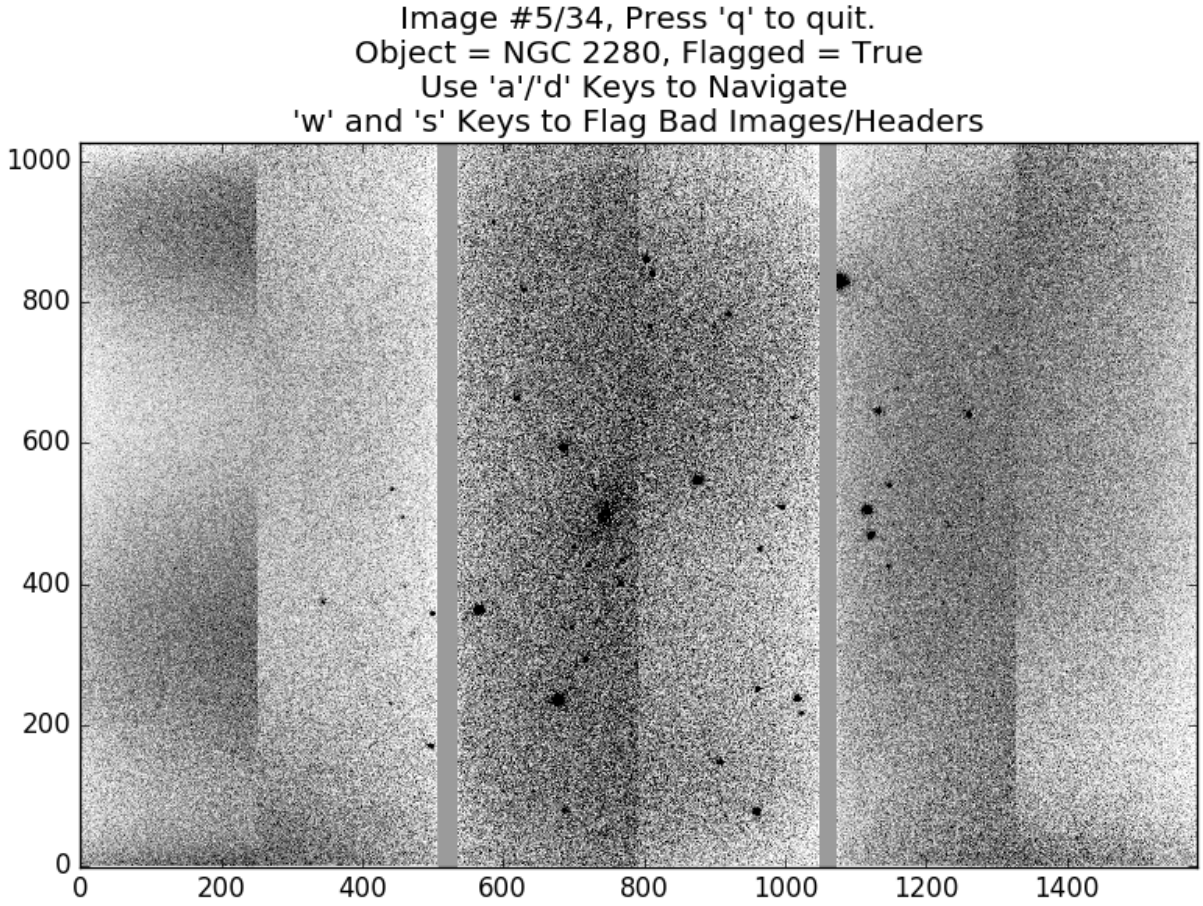
This step in the reduction process creates a correctly aligned mosaic of SALT RSS's 6 CCD chips, performs crosstalk/gain/bias corrections, and does cosmic ray masking/removal. It also creates additional image extensions in each image file for keeping track of the noise and bad pixel masks.

Note: I've noticed that occasionally the *zSALT* package gets stuck and hangs forever on my machine during the cosmic ray removal process. If you're encountering this problem, locate `zsalt/imred.py` and disable multithreading by changing line 103 to read `multithread=False`.

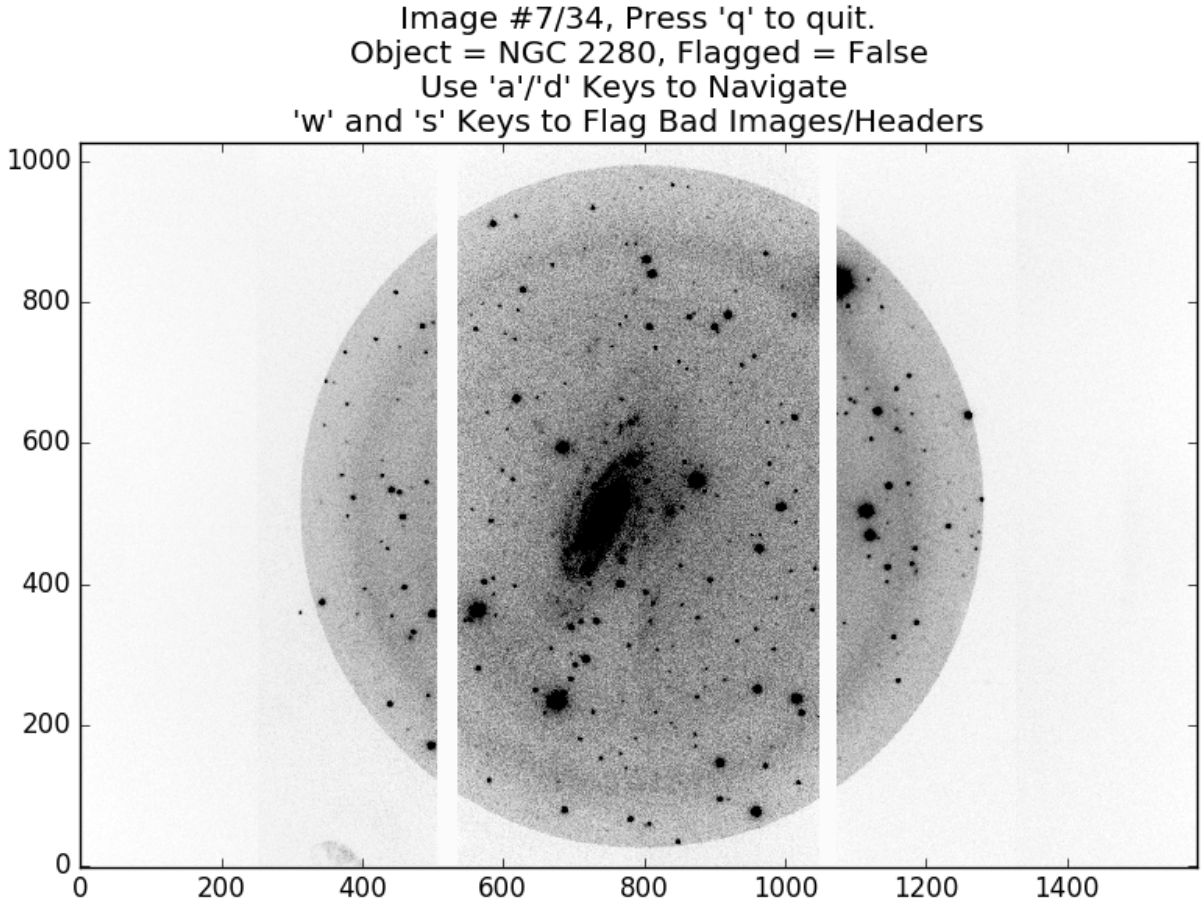
1.2.3 Image Verification

Next, the pipeline will display each image in your 'product/' directory and ask you to verify that the image and its header are correct. This step is not strictly required, but I highly recommend it. Relatively often, acquisition images of your object may be incorrectly marked as 'ARC' images in their headers. Or images of Neon/Argon calibration lamps will be marked as 'FLAT' in the headers.

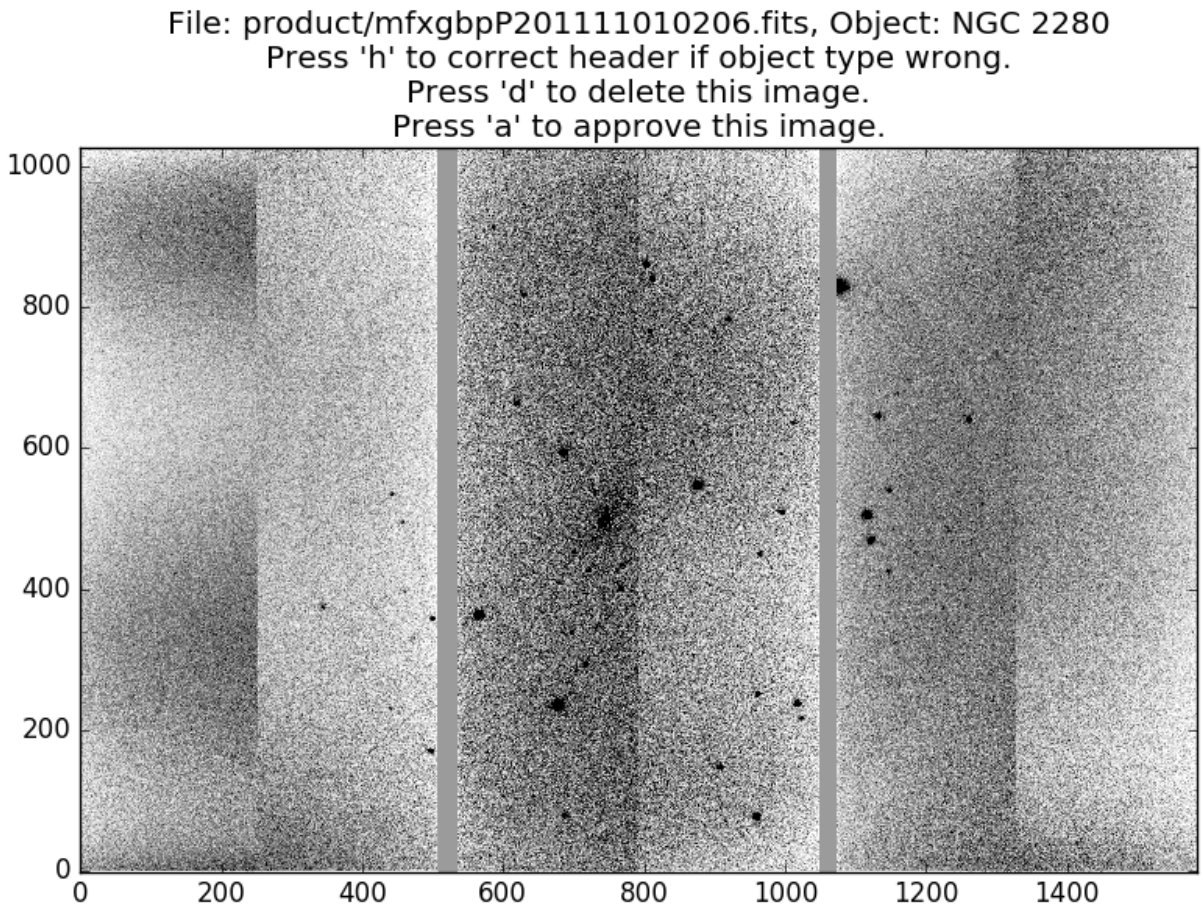
When in the interactive plotting environment, use the 'W' key to flag an image as bad, or the 'S' key to unflag an image. Use the 'D' and 'A' keys to navigate forward and backward between images.



This image is an acquisition image, but is marked as 'NGC 2280' in its image headers. I don't want to use this image, so I've used the 'W' key to flag it as bad. Note the 'Flagged=True' in the plot title.



This image of NGC 2280 is good and its header information appears correct. Because it's good, I haven't flagged it. Once the flagging process is completed, any flagged images reappear for a review process.

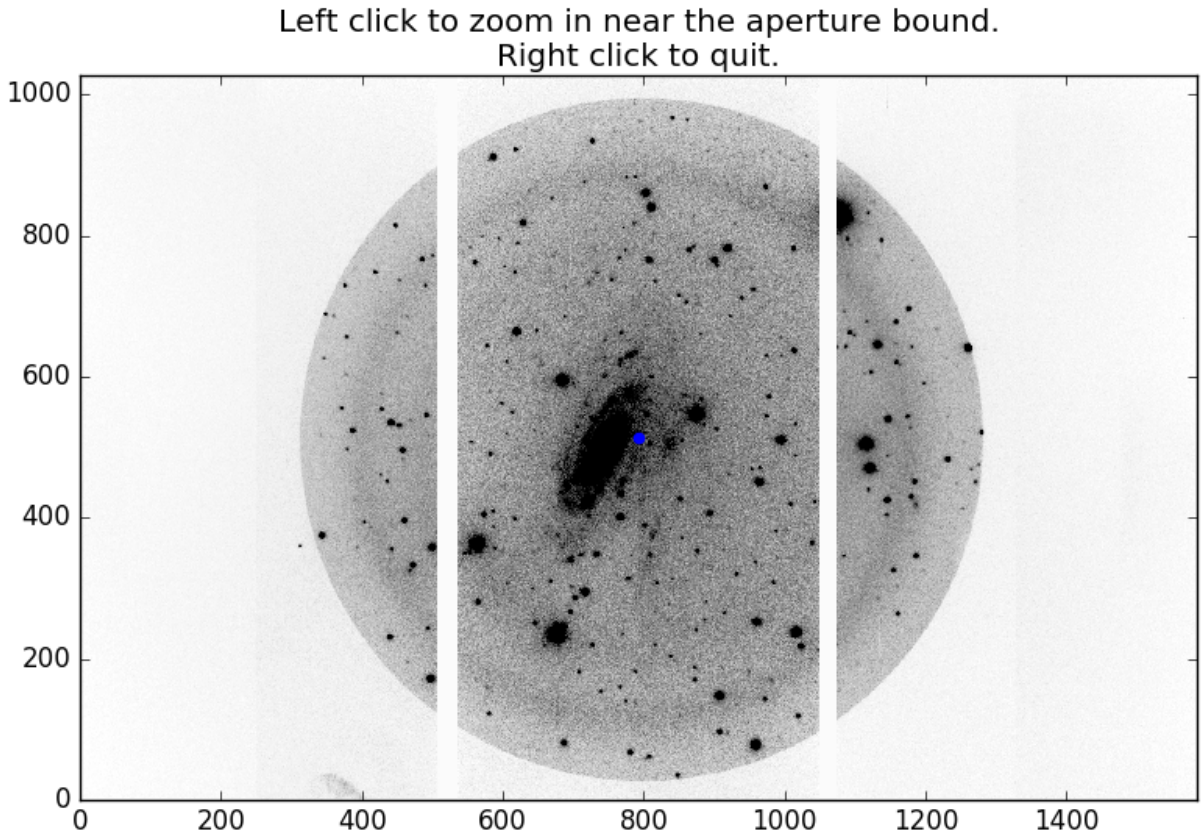


Here's the acquisition image again. I don't want to use this image, so I press 'D' to delete it. If its header information had been incorrect, I could have pressed 'H' to input new header information. If I had flagged it by mistake, I could have pressed 'A' to approve the image.

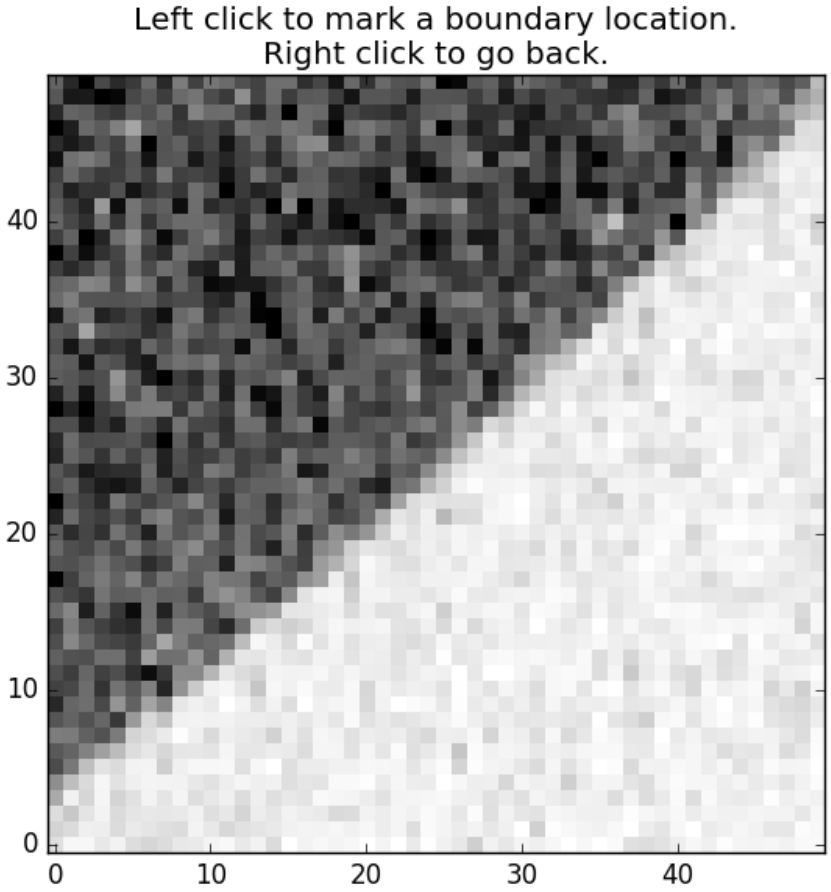
Note: Pressing 'D' doesn't actually delete files. Rather, it changes a keyword in their headers called 'FPGOOD' to False. If you mistakenly 'delete' an image during this process, you can manually edit this header keyword back to True to undo your mistake.

1.2.4 Aperture Masking

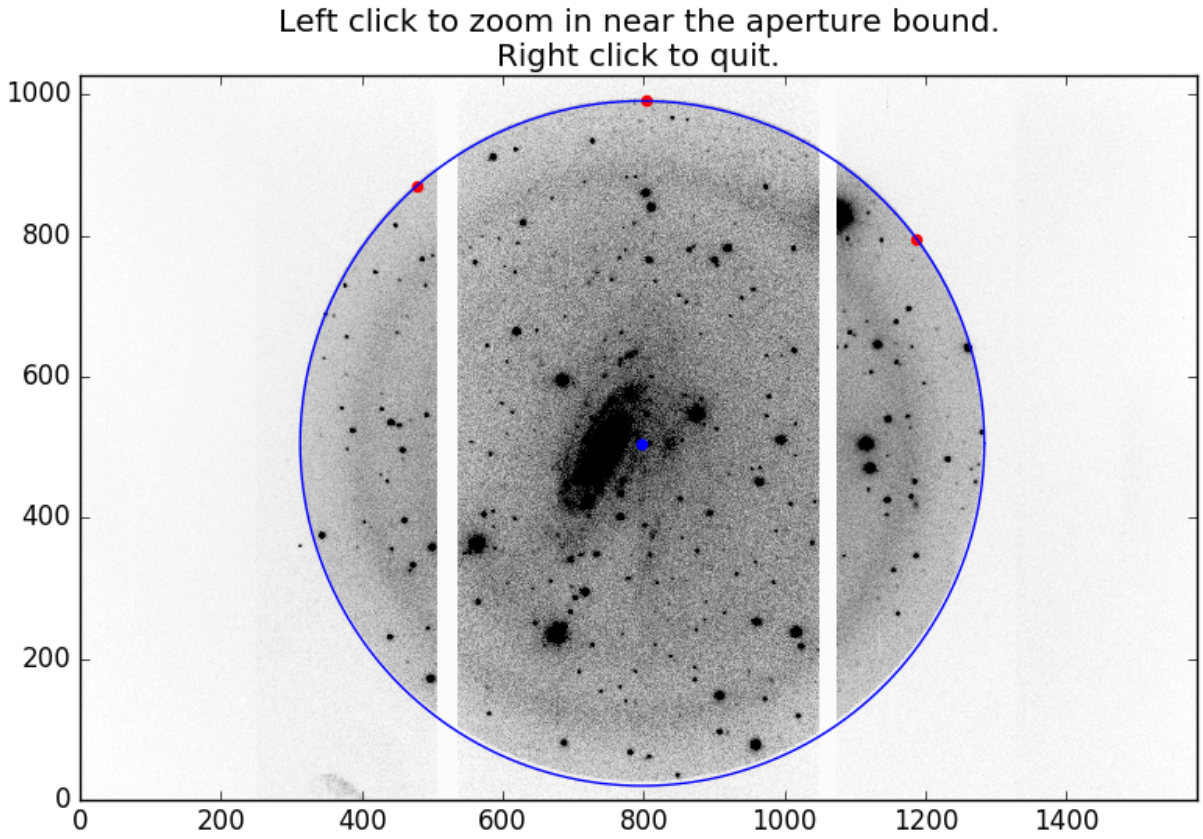
SALT FP images are taken with a circular aperture mask in place. This next step in the pipeline adds pixels outside this circular aperture to the bad pixel mask. To create the circular aperture, mark a few points near the aperture boundary by clicking on them with the mouse.



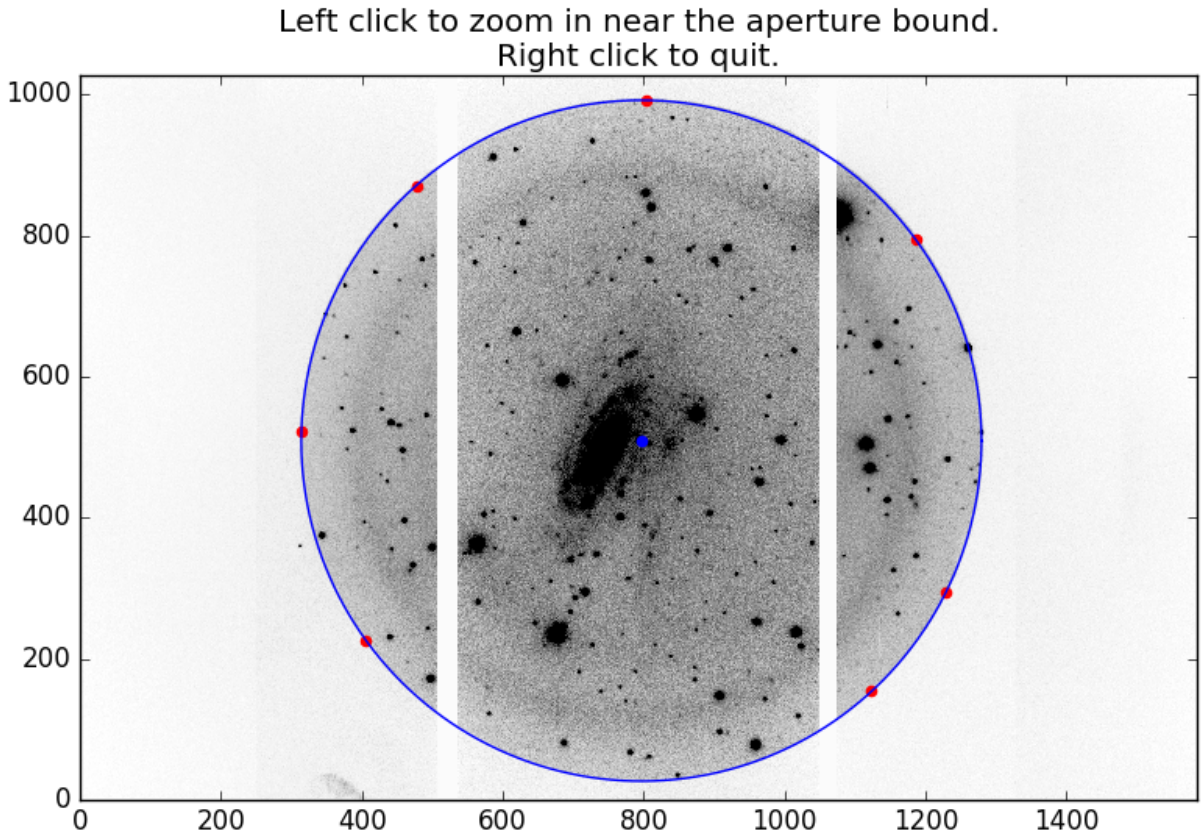
Here is one of my images of NGC 2280 before I've marked any points along the boundary. I decide to left click on a point near the bottom right of the image to zoom in on that region.



Here's a zoomed-in view of the area near where I clicked. I can left click again to mark a point on the boundary, or I can right click to zoom back out.



Now that I've marked three points along the boundary, the code tries to fit a circle to those points. Note that at the bottom of the image, the fitted circle does not align with the aperture boundary. Because the points I've marked are all on one side of the image, the circle does not provide a good fit to the aperture.



After marking a few more points near the aperture all the way around the image, I now have a good fit to the aperture boundary. Now I right click with my mouse to exit the interactive view.

The pipeline will now add any pixels outside this fitted circle to the bad pixel masks of every image in the 'product/' directory.

1.2.5 Adding DS9 Region Files to the Bad Pixel Mask

Occasionally, you will want to mask a portion of your image that was affected by a satellite trail or some other unexpected phenomenon. I have added support for this using DS9 region files. If you have an image 'product/image_name.fits' and you wish to mask a geometric portion of it, create a region file called 'image_name.reg' and place it in the directory *above* the product directory. In that region file you may have any number of DS9 regions, one per line.

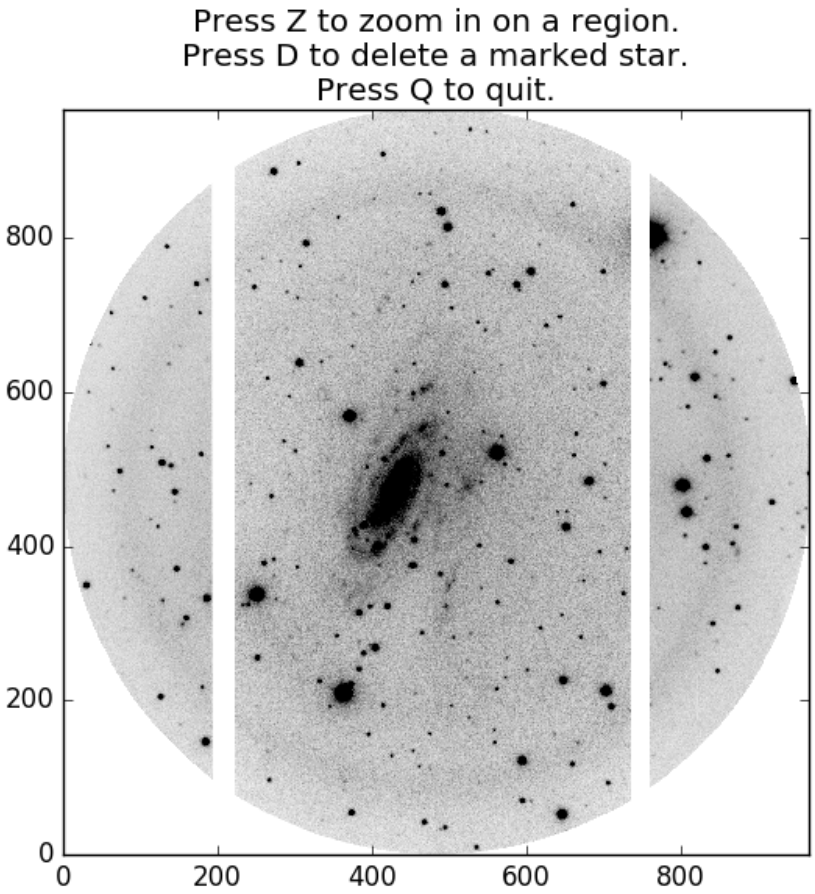
The supported shapes are 'Circle', 'Box', and 'Ellipse'.

The supported file formats are 'DS9/Funtools', 'IRAF PROS', 'CIAO', 'SAOimage', and 'SAOtng'.

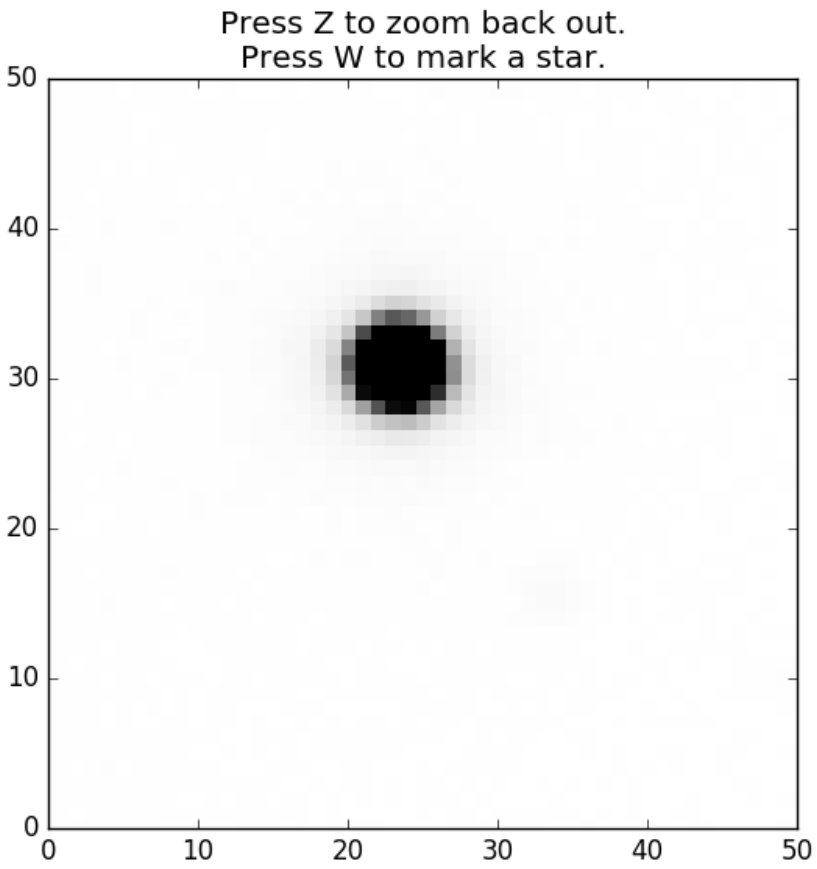
When saving the region file, **always use the image coordinate system**; not physical, WCS, or any other.

1.2.6 Measuring Seeing FWHMs

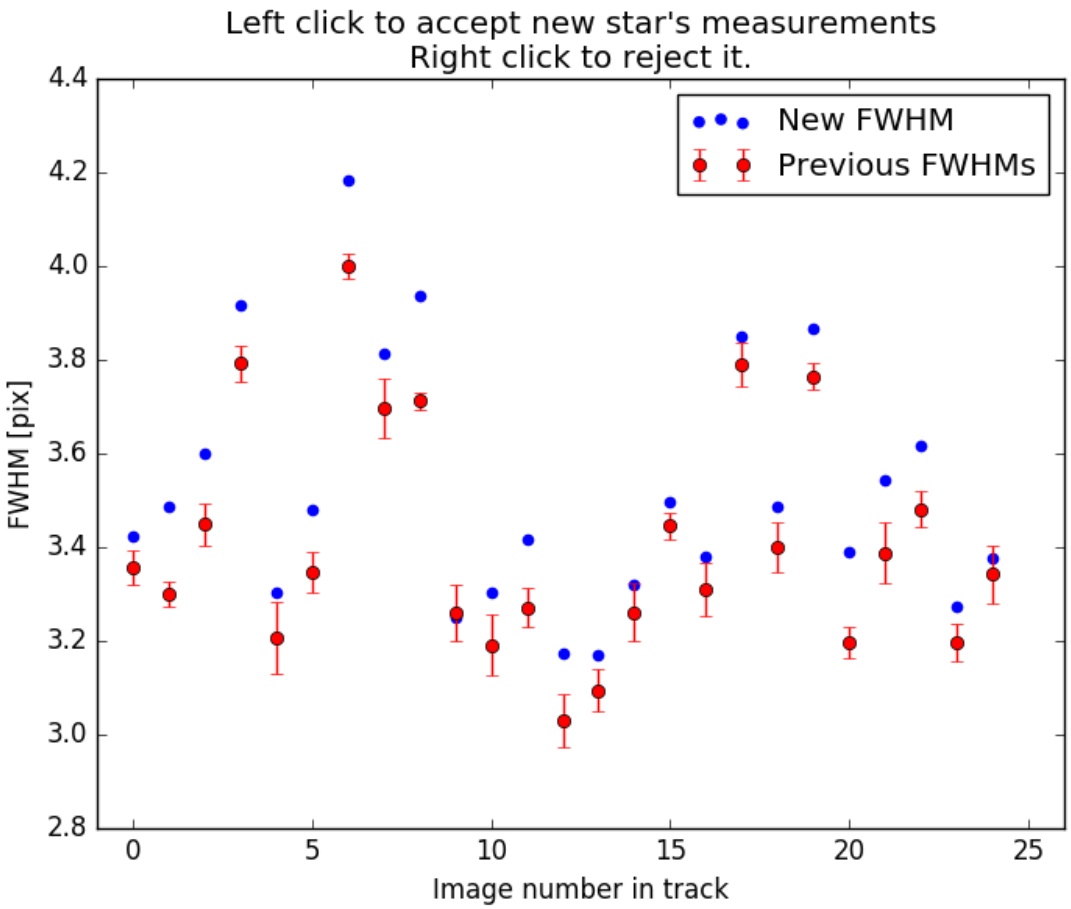
Next, the pipeline needs to know the typical full-width at half-max (FWHM) of point sources in your images, also known as the point spread function, or PSF.



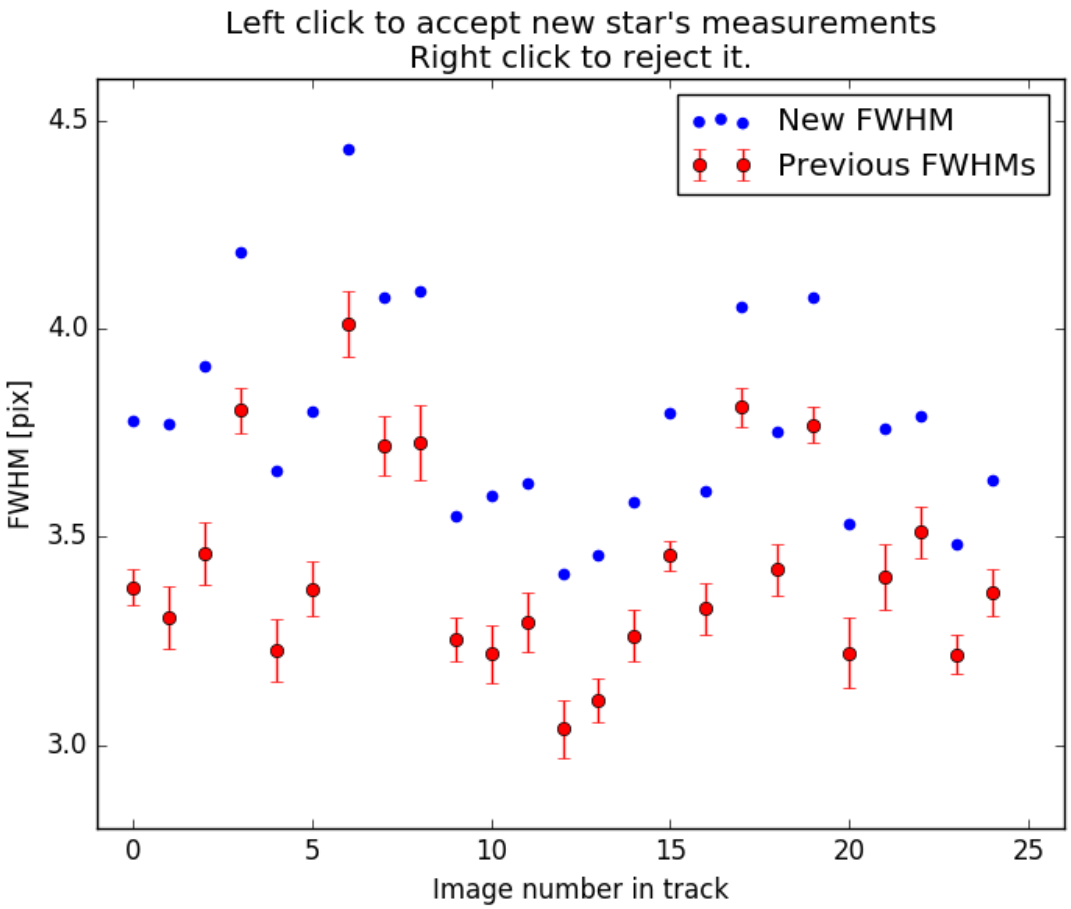
Another interactive plotting window will appear, this time asking you to press 'Z' to zoom in on a star. Find a star in your image and place your mouse cursor over it, then press 'Z'.



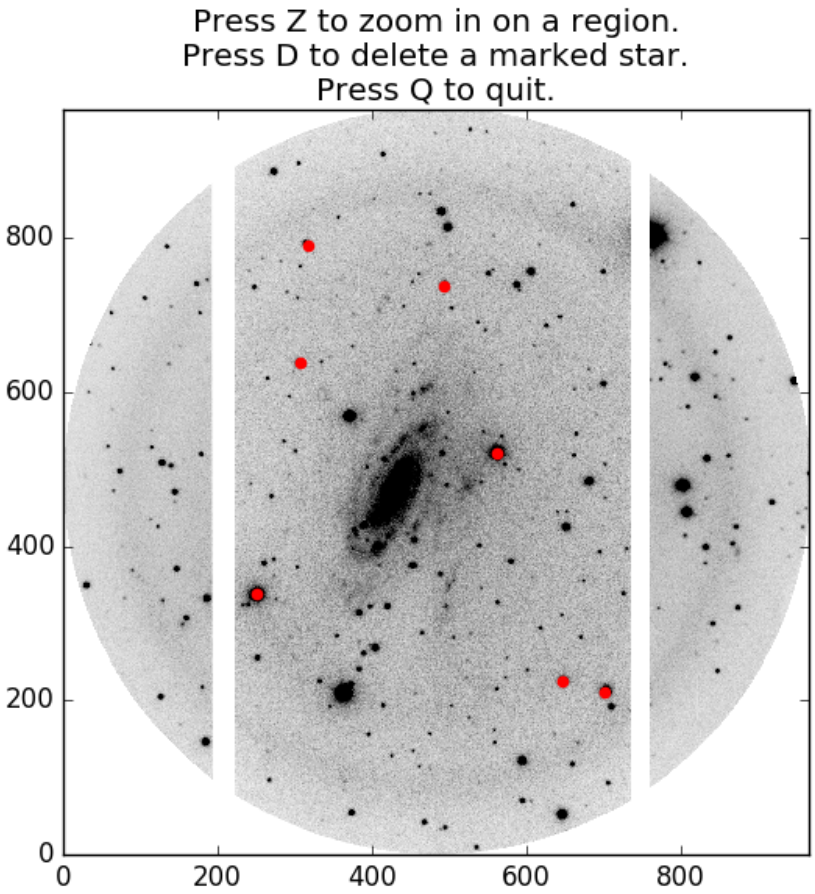
A zoomed-in plot of the area around this star appears. Again move your cursor over the star, then press 'W' to fit for the star's FWHM. Alternatively, if you don't like this star, you can press 'Z' to zoom back out.



The code will now fit a Gaussian profile to that star in each of your images, then display the FWHM as a function of image number. The mean and standard deviation of stars you've previously fitted will be displayed in red. The newly marked star will be displayed in blue. To approve the newly marked and fitted star, left click on this plot. To reject it, right click. The newly marked star in the above plot seems to agree reasonably well with my previously marked stars, so I left click to approve it.



This star seems to be systematically blurrier than the several other stars that I've marked, so I right click to reject the fit.

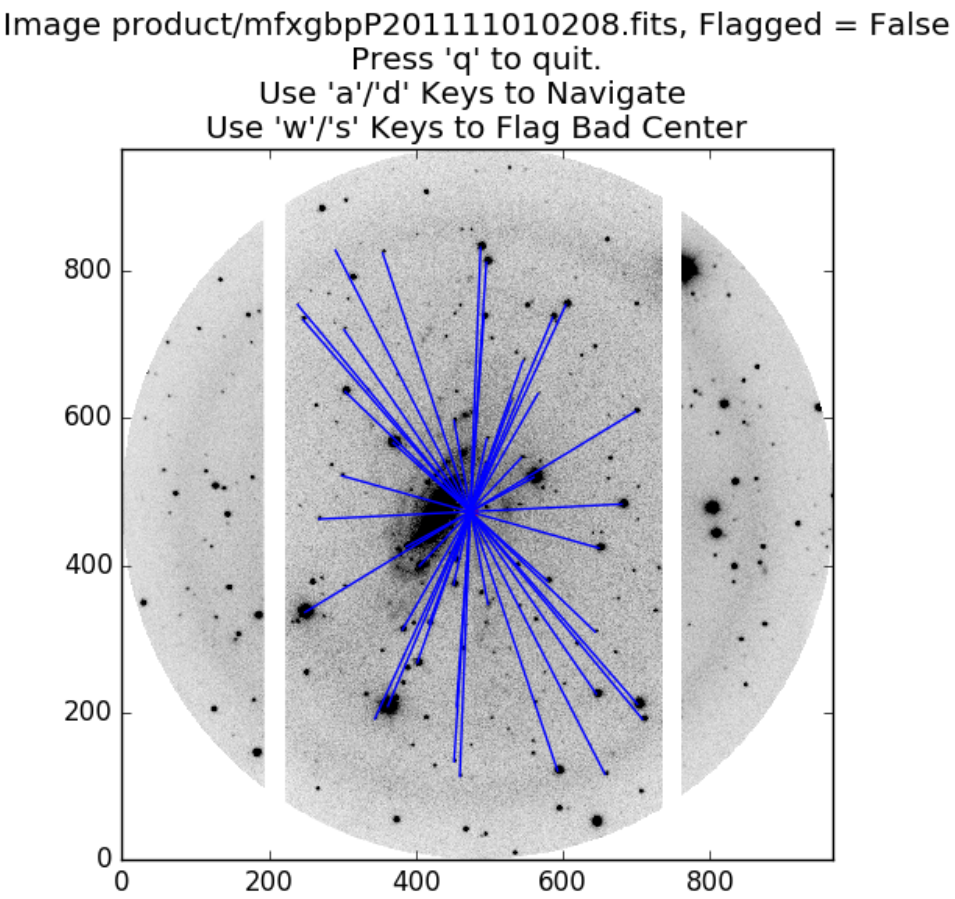


Once I've marked several stars and have a fit I'm happy with, I press 'Q' to quit the interactive plotting window. Alternatively, I can use the 'D' key to delete a star I've previously marked.

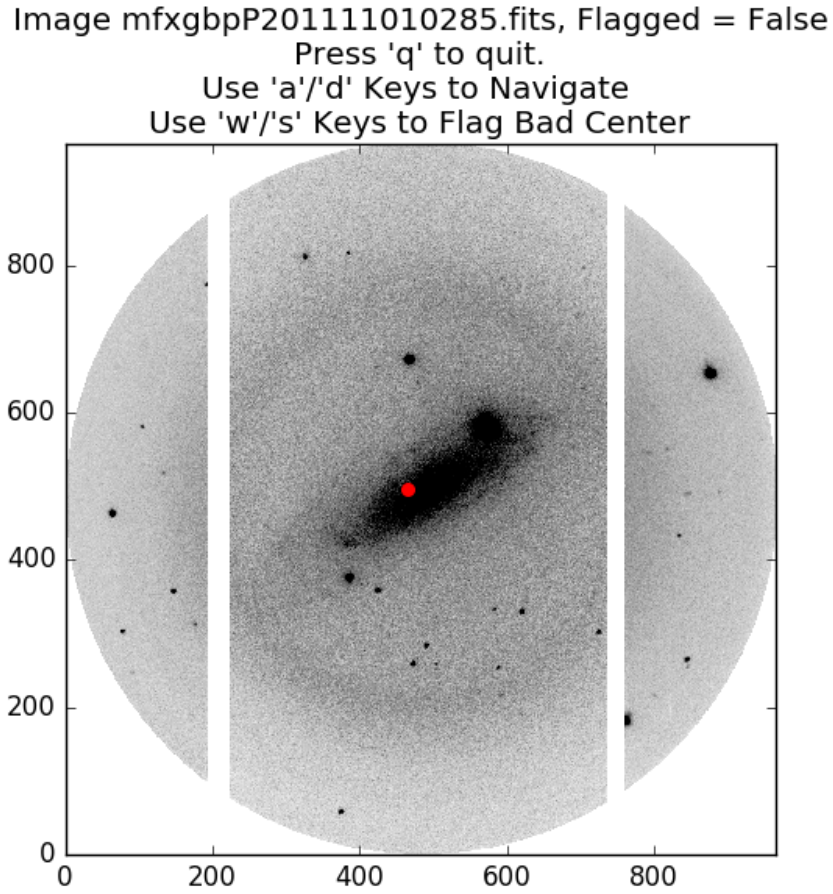
After the interactive marking of stars is complete, the mean FWHMs are written to the image heads under the keyword 'FPFWHM'

1.2.7 Ghost Center Fitting

Each bright object in a SALT FP image will have a reflection of itself appear at the other side of the image, reflected about a common center. This center's location is very important later in the data reduction process for finding the wavelength solution. This next part of the pipeline fits for this center by trying to identify star/ghost pairs in the images. In dense star fields, this works quite well. In sparse fields, this routine tends to require some personal intervention.



In the dense star field around NGC 2280, the routine has identified dozens of star/ghost pairs and very accurately found the center. I press 'Q' to approve this fit and move on to the ghost subtraction section.



In this sparse star field around NGC 1325, the routine has failed to find the ghost center, so I press 'W' to flag the center as bad. The pipeline will not proceed without a good center fit. When this happens, it's best to try doing things manually. For that, use the find_ghost_centers function.

1.2.8 Flat-fielding

If you have any images with the object type 'FLAT' in their image headers, this routine creates a combined flatfield image from them. The combined flatfield image is placed in the current directory (i.e. where you ran the pipeline code) as a file 'flat.fits'.

The pipeline then searches for this 'flat.fits' file and uses it to flatten each of your images. If the pipeline cannot locate the 'flat.fits' file (perhaps because you had no flatfield images in your 'product/' directory), you will be prompted to enter a path to a flatfield image elsewhere on your filesystem.

You can also skip the flattening step by leaving this prompt blank. This is not a great idea in general, but can sometimes be the only option if you don't have suitable flatfield images.

Occasionally you may wish to have a bit more control over the flatfielding process (e.g. flattening each data image by a different flatfield image). For this, use the flatten function.

1.2.9 Object Directory

At this point, the pipeline creates a new directory named after the object you've observed and copies all images associated with that object to the new directory For the examples I've been using, the pipeline creates a directory

called 'NGC2280'.

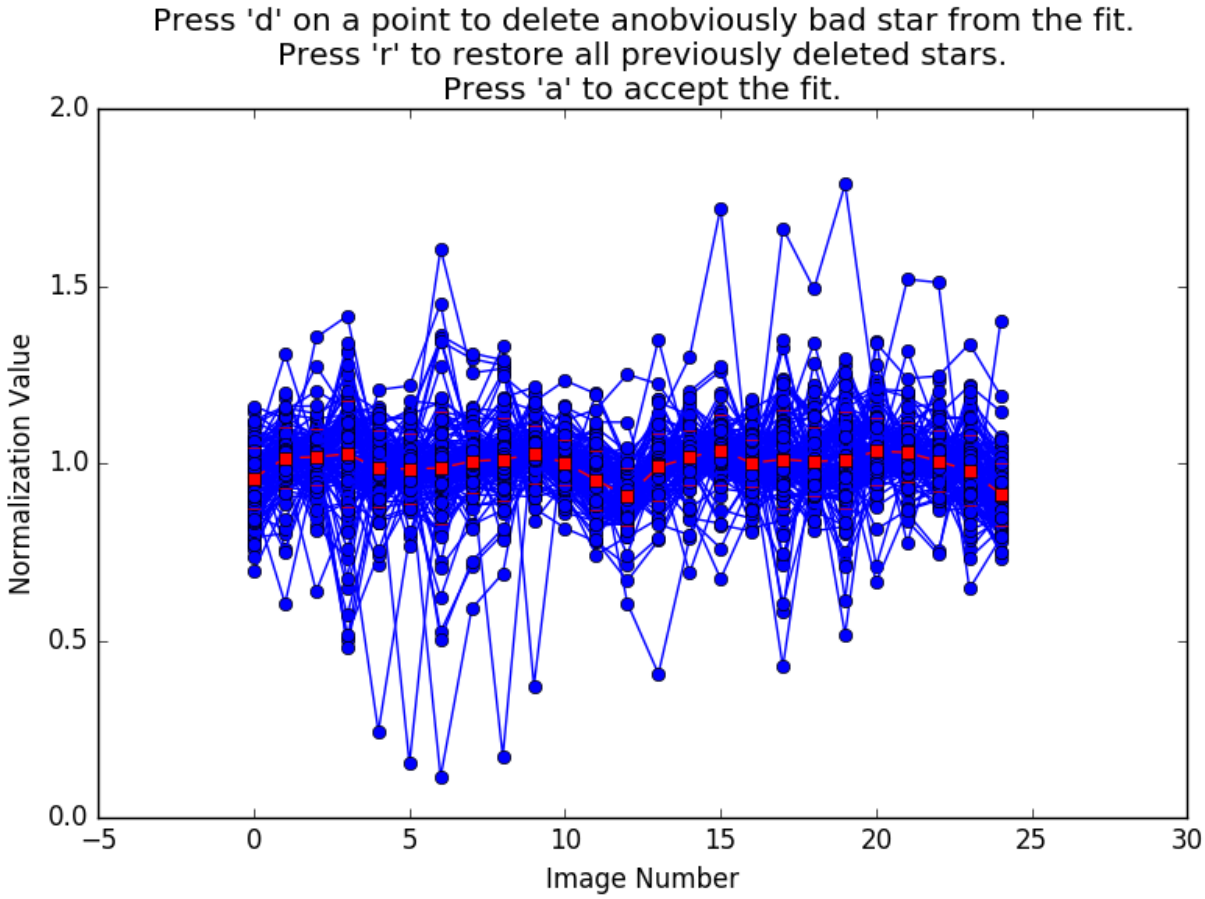
The purpose of this is to create a 'backup' point before proceeding with the rest of the routines. Some of these routines are more prone to error, and this preserves a "pristine" version of the image files in the 'product/' directory.

1.2.10 Image Normalization and Alignment

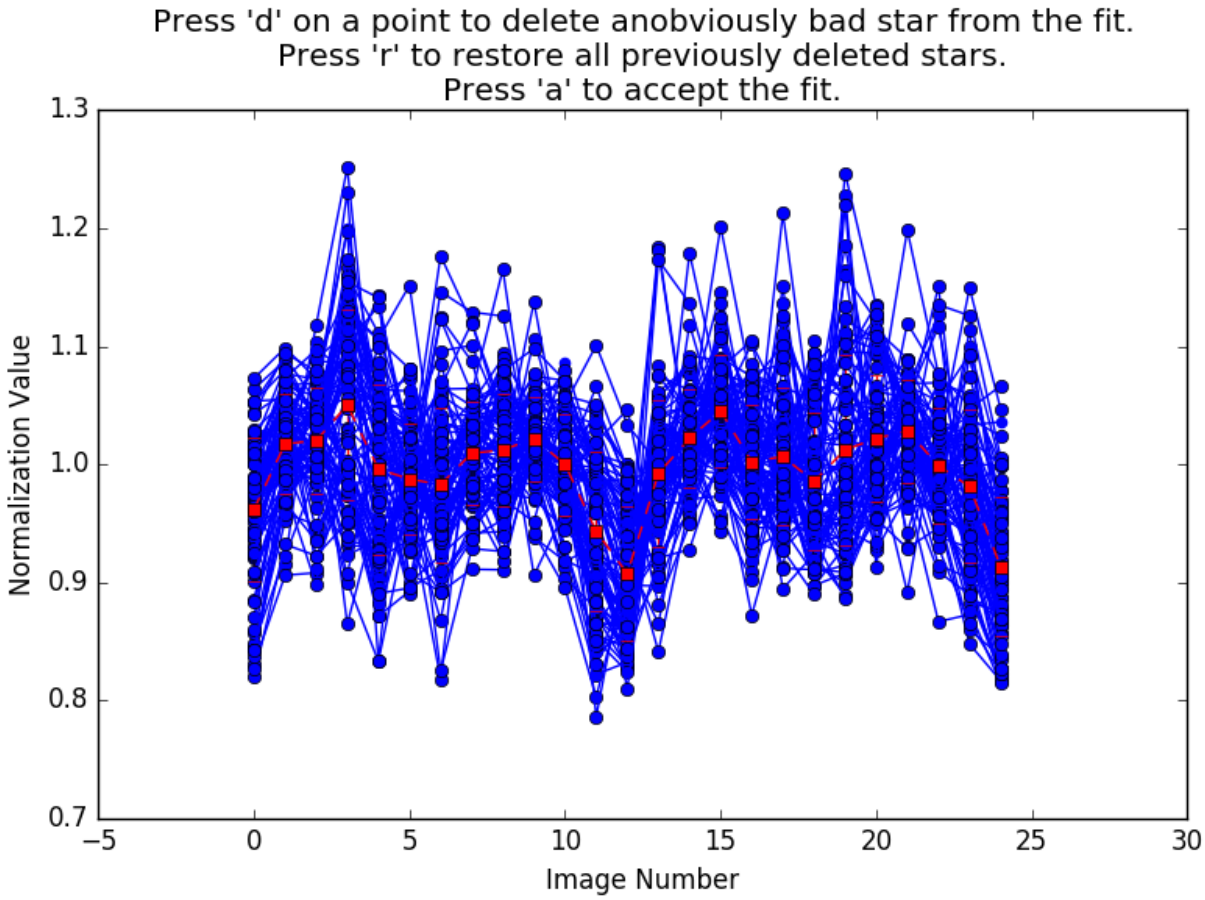
This routine attempts to accomplish two things simultaneously:

- Determine the relative normalization of your images to correct for SALT's "moving pupil" design.
- Determine the relative shifts in X and Y necessary to align all of your images to a single coordinate system.

It accomplishes this by detecting the stars in each image and attempting to match them across all of your images. For each star, the star's relative intensity is then plotted as a function of image number in your observation:



Each detected star is plotted as blue points connected by lines. The average and standard deviation for each individual image is plotted in red. If you have any stars which are clear outliers from this distribution, you can remove them by moving your mouse cursor over them and pressing the 'D' key. The star detection algorithm is pretty liberal, so you're likely to have a few of these outliers in almost any observation. In the above plot of the stars near NGC 2280, there are several stars which are pretty severe outliers in a few of the images, so I press 'D' repeatedly to remove those stars.



After removing several stars from the fit, I'm left with what looks like a pretty reasonable distribution of stars. When you're finished, you press 'A' to approve the remaining distribution.

Note: The Y-limits of this plot automatically rescale as you remove stars from the fit. While it seems like scatter might be increasing as you remove stars, that's merely a consequence of the Y-limits getting tighter.

The pipeline will then calculate a normalization value and uncertainty in this value for each image and rescale that image and its variance appropriately. The average coordinate shifts of these stars are also recorded in the image header keywords 'fpxshift' and 'fpyshift' for later use.

If the pipeline is having trouble detecting enough stars, or is detecting too many stars, or is having trouble matching stars across images, you should consider using the align_norm function manually.

1.2.11 Making a Median Image

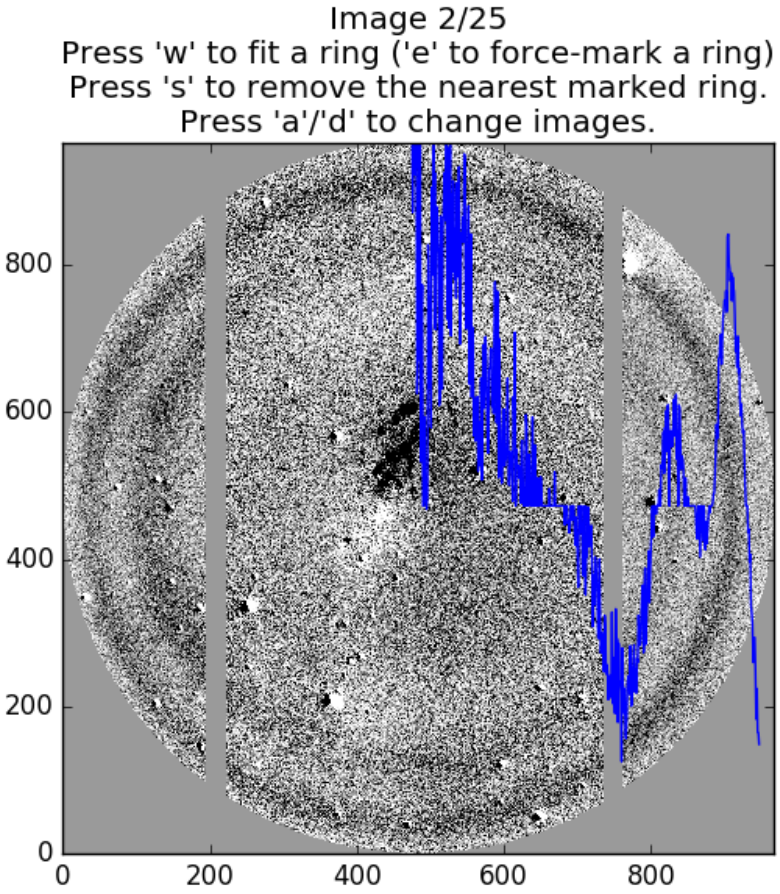
Once the images are normalized to one another, the pipeline produces a file 'median.fits', which is a median-combined image of all of the data images in an observation. The median image is useful because any qualities which differ from one image to another (e.g. emission line rings, galaxy emission, cosmic rays, satellite trails) are washed away in the process of taking the median. This process is pretty fool-proof, and shouldn't require any intervention.

1.2.12 Wavelength Calibration

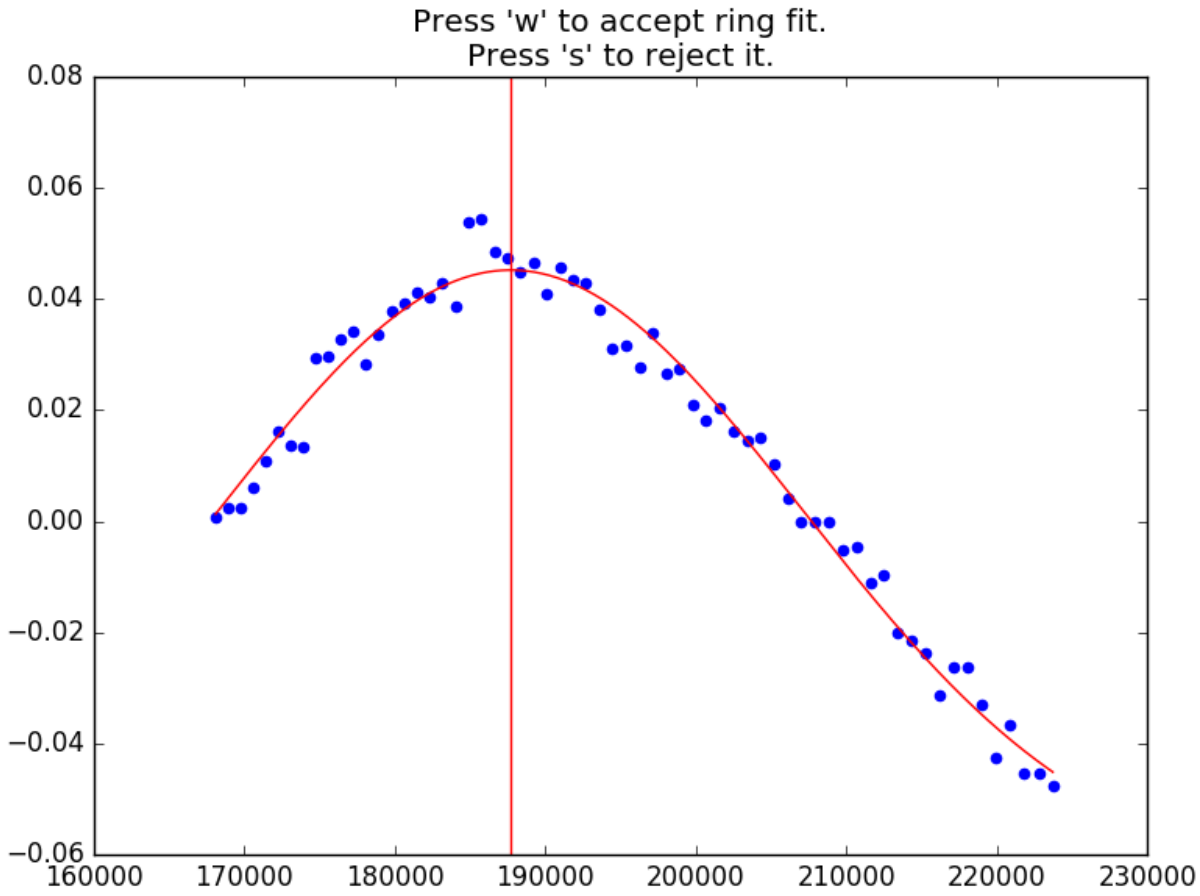
Note: Currently, only the 'PI06530' and 'PI06645' filters have their Night Sky and ARC Calibration wavelengths recorded as libraries in the pipeline. If the filter you're using isn't supported, please consider contributing one to the project.

Wavelength calibration is where things start to get tricky. I've found that the best measure of the wavelength solution in SALT FP data are the night sky emission line rings which appear in the data images themselves. These rings are symmetric about the same center as the ghost reflections and arise from rotational transitions of molecules in Earth's atmosphere.

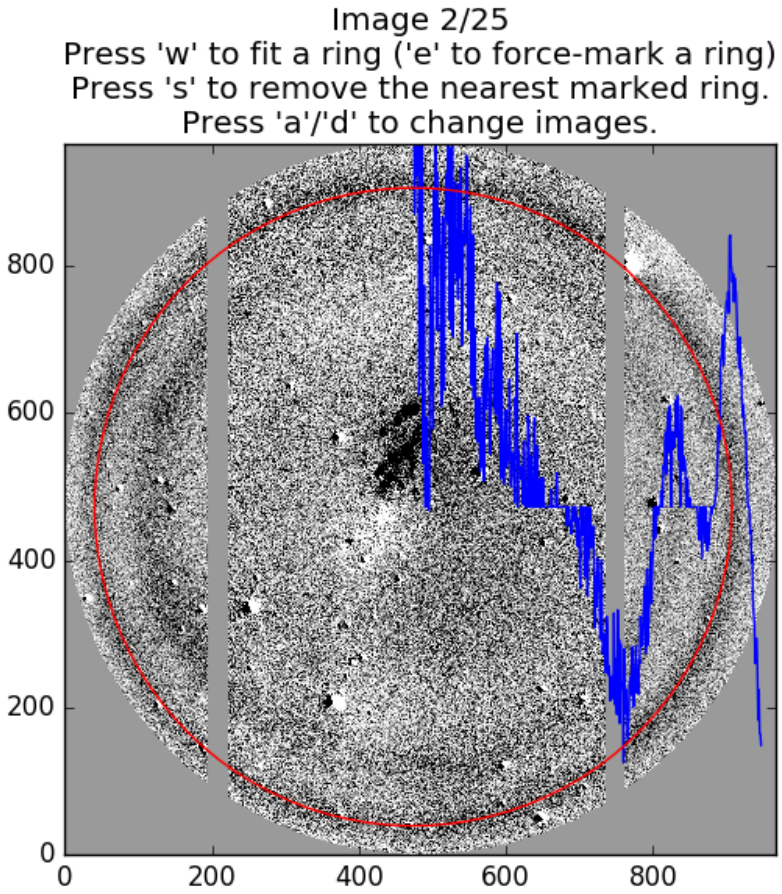
The pipeline will subtract the median image from the previous step from each image of your object. Ideally, this results in an image which is a combination of only background noise and excess emission above the median level. The subtracted image is plotted in an interactive plotting window, along with an azimuthally-averaged radial profile of the excess emission.



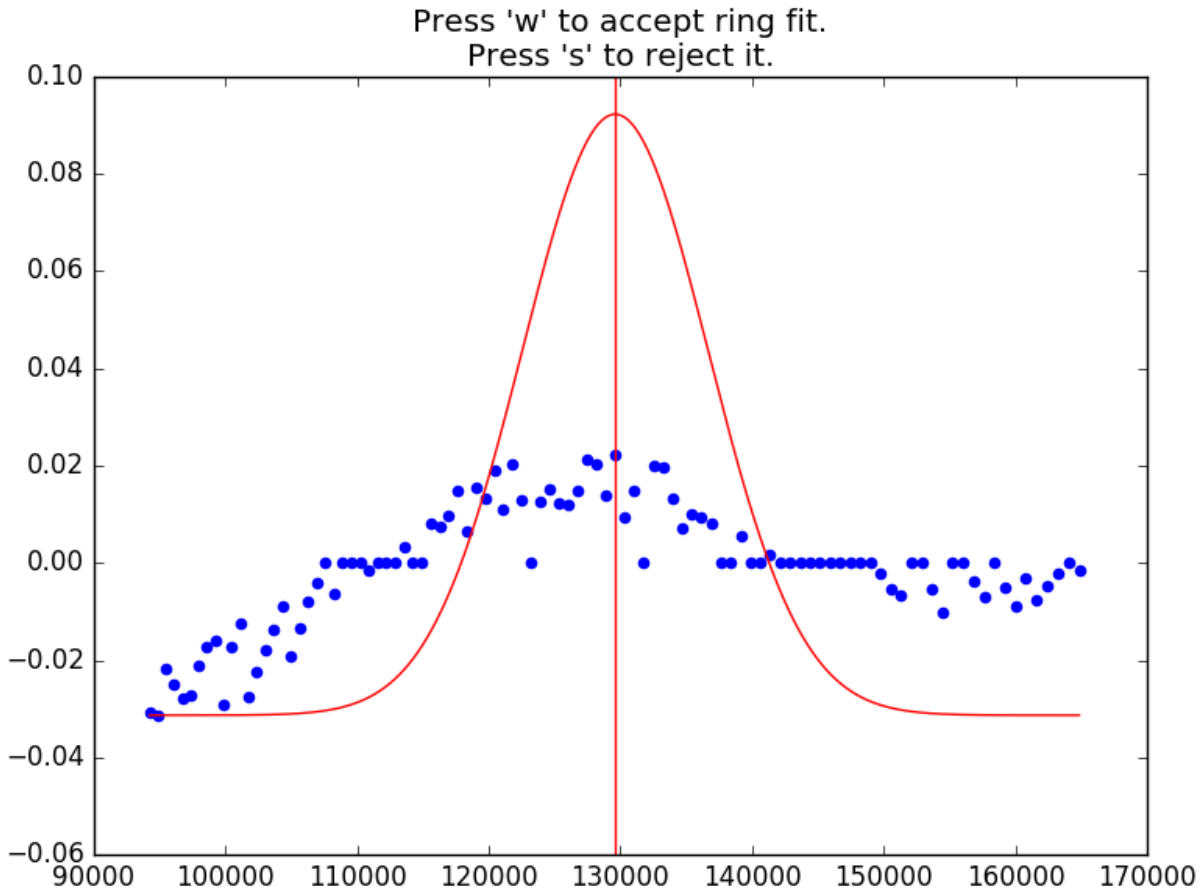
Here is an image which has two night sky emission line peaks (one is considerably brighter than the other). By hovering over a peak in the blue radial intensity profile with the mouse and pressing the 'W' key, you can mark an emission line ring.



When 'W' is pressed over a ring, the code attempts to fit a small radial range with a Gaussian function and displays the fitting attempt. If you're happy with the fit (as I would be with the above fit), pressing 'W' again will approve it. Pressing 'S' will discard the ring.

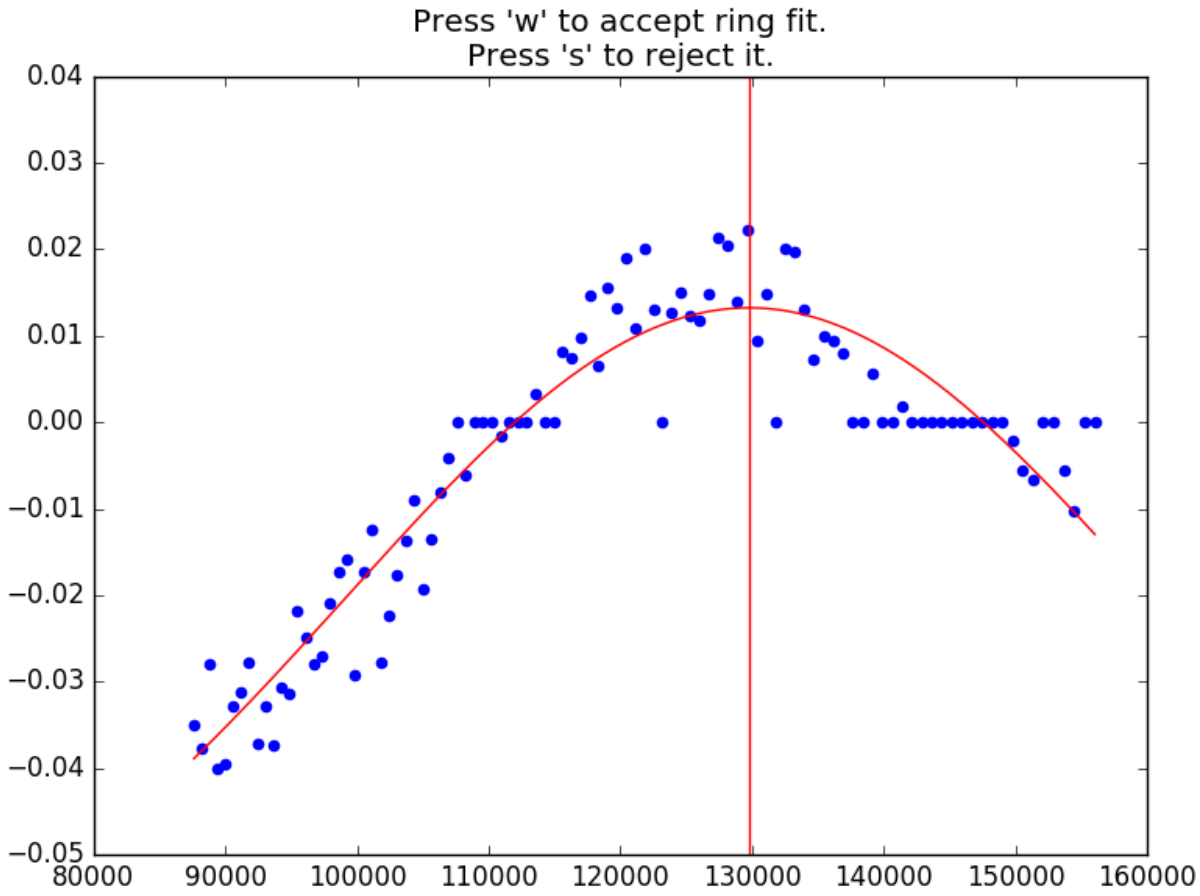


Note that the ring I just marked is highlighted by a red circle. Now we want to fit the other ring. Again, I press 'W' while hovering over the peak.

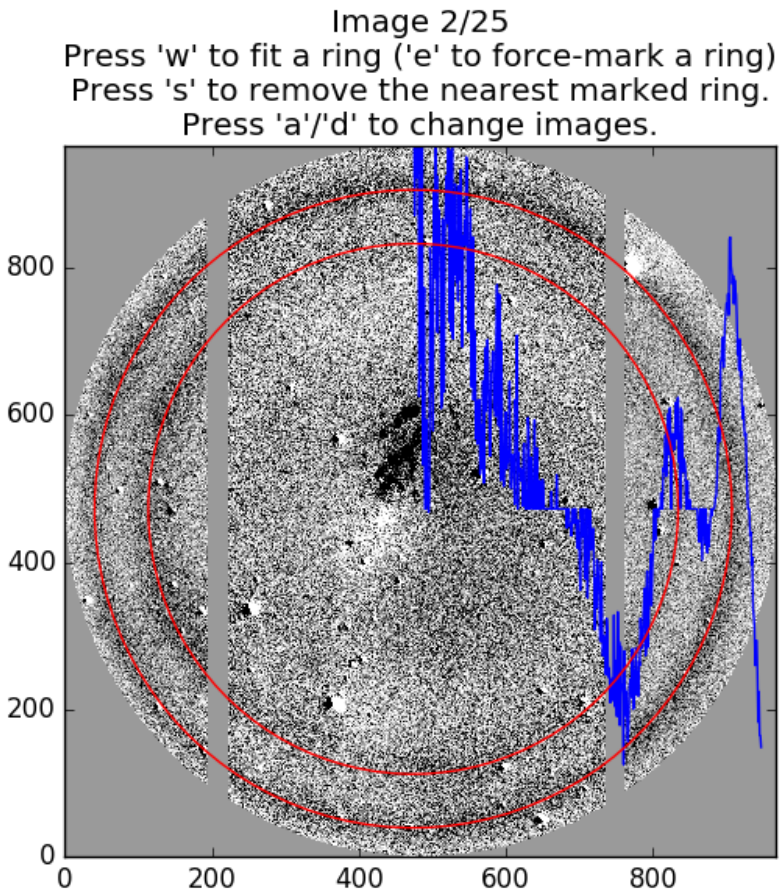


This attempt at fitting the emission line clearly didn't go so well, so I press 'S' to reject it.

Let's try again by pressing 'W' in a slightly different position near the emission peak.



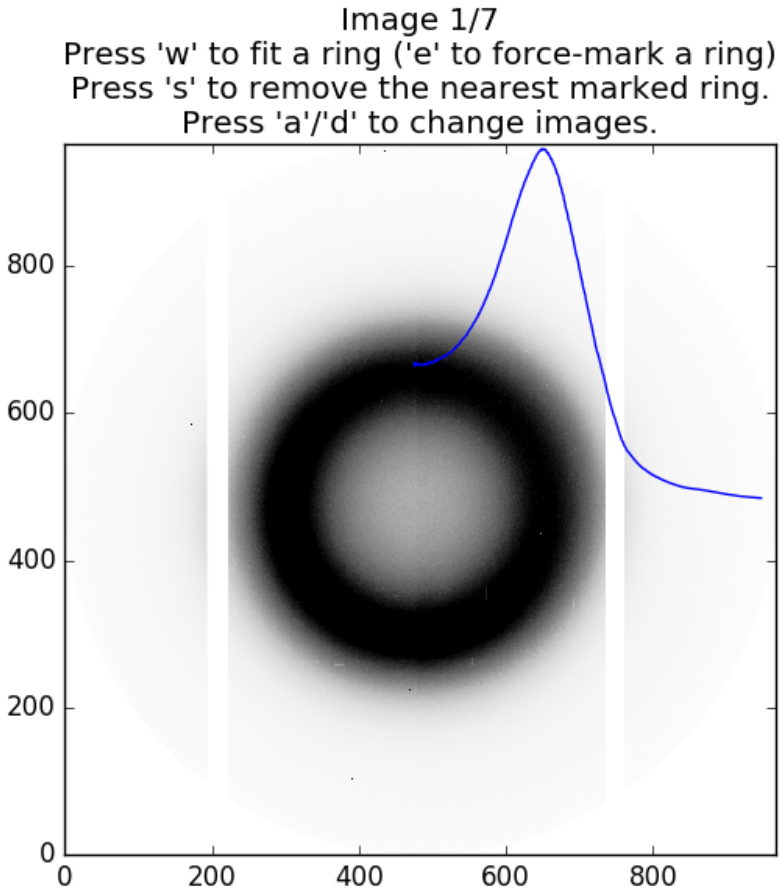
This looks much better, so I press ‘W’ to accept the ring.



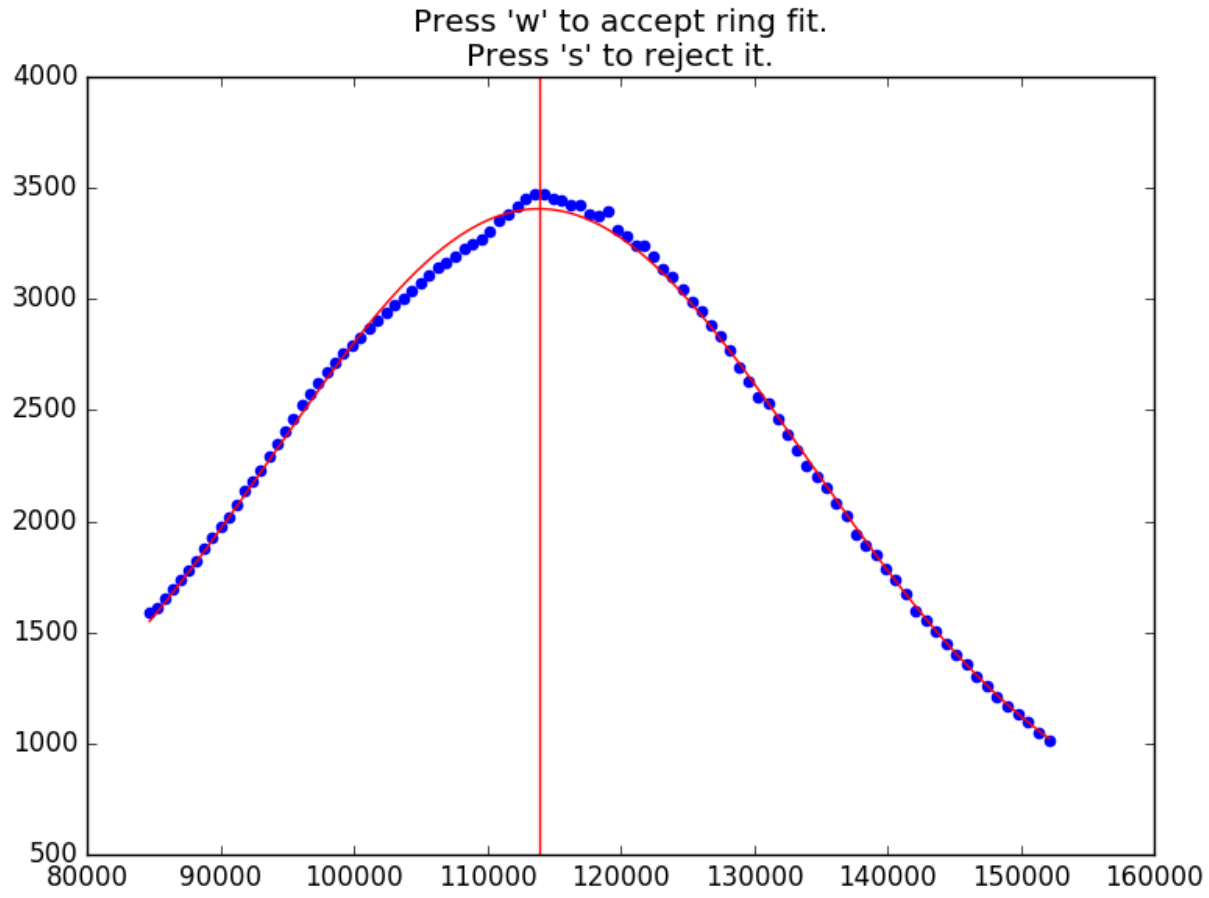
Now that both rings are marked, I press 'D' to move to the next image.

Note: If you're really having trouble getting the Gaussian fitting routine to fit a ring, you can manually force it by pressing 'E' while hovering your cursor over the ring. Don't do this unless you're really confident in your ring.

After repeating this process in all of your data images (which can be quite arduous), you'll perform a similar procedure with any ARC calibration images you have:

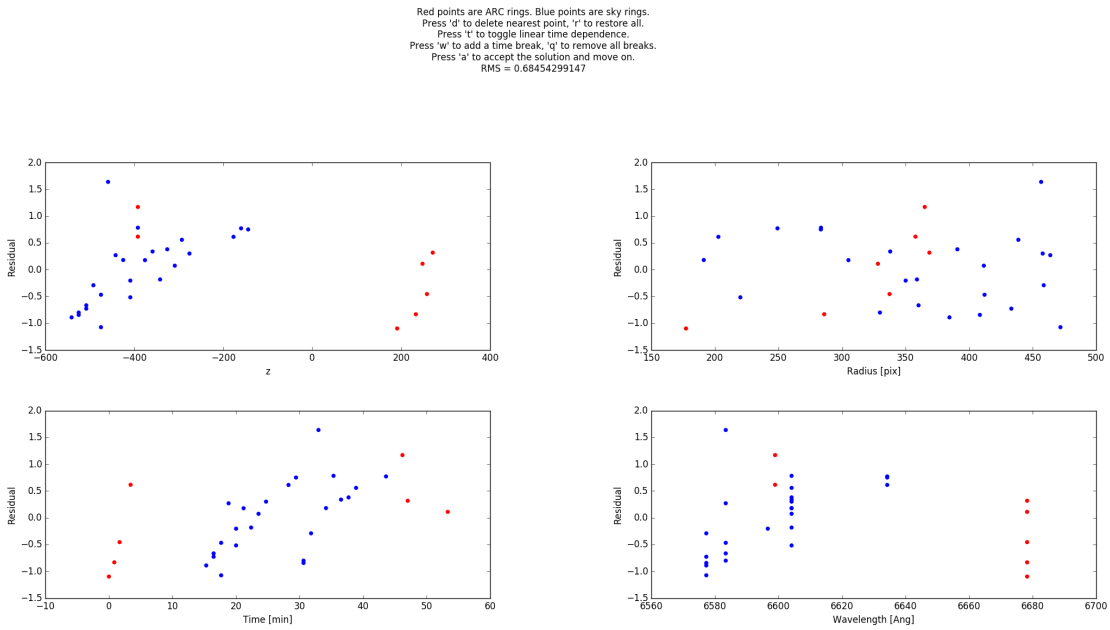


Again, press ‘W’ while hovering over a ring peak to fit the peak with a Gaussian.



The symmetry centers of the ARC images are not as well known as the data images (due to the lack of ghosts), so sometimes the plots will look skewed or double-peaked like this. In my experience, this is okay. Again, press 'W' or 'S' to approve or reject the ring fit, then press 'D' to move on to the next image.

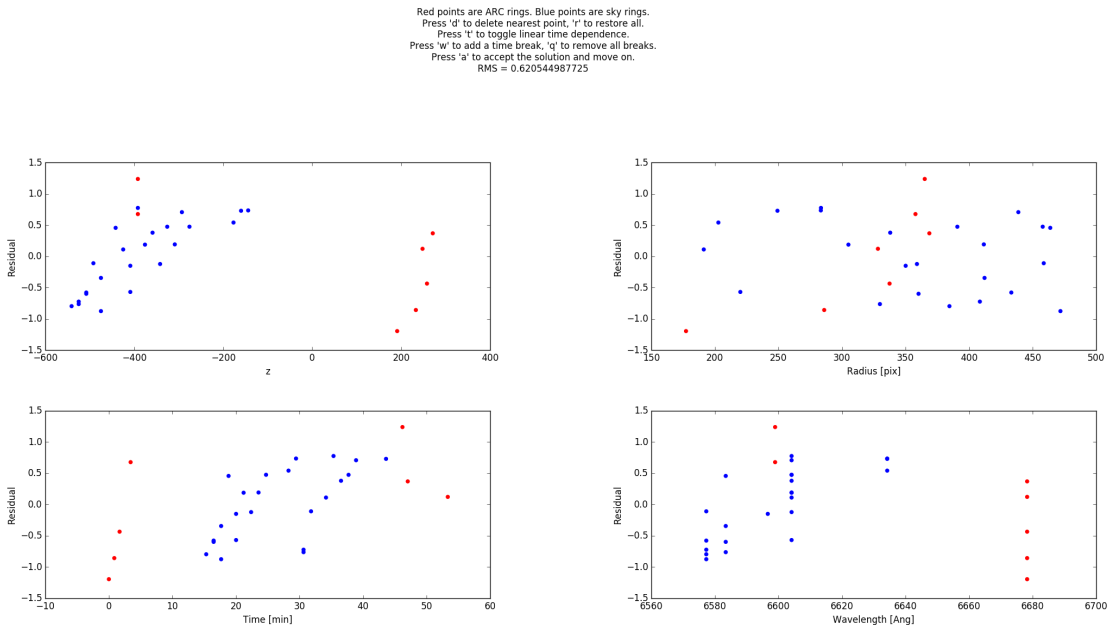
After repeating this process with all of your ARC images, a new interactive plot window will appear:



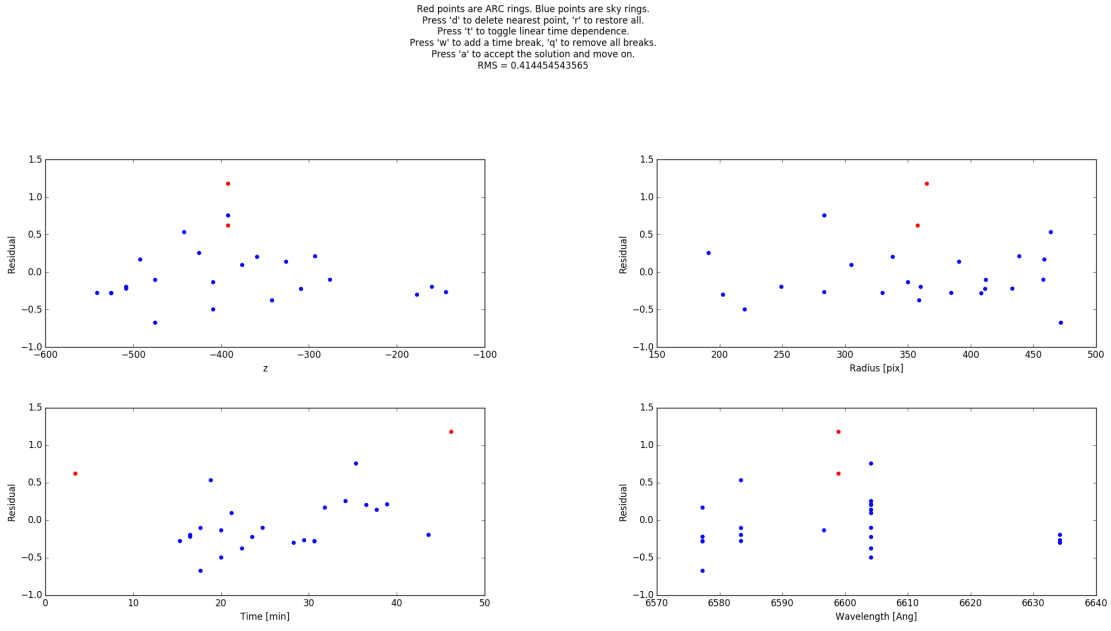
This plot has **a lot** of information on it and it's worth discussing it in detail here. For each of the rings you've marked, the pipeline decides on which wavelength the ring most likely corresponds to. It then performs a least squares regression to the rings you've marked, trying to calculate a wavelength solution which matches all of your rings as best it can. The functional fit is of the form `lamdba = (A + B*z + E*t)/(1 + r^2/F^2)^0.5` By default, E is set to zero but can be toggled on by pressing 'T'. The RMS residual to the fit is displayed at the top of the plot. Lower values are generally better. The blue points on these plots correspond to night sky emission lines while the red points correspond to ARC calibration lines. The four plots show the residual to this fit as a function of:

- Top Left: `z`, the control paramter which determines etalon spacing.
- Top Right: `r`, the radius in your images
- Bottom Left: `t`, the time of the observation
- Bottom Right: `lambda`, the wavelength of the ring.

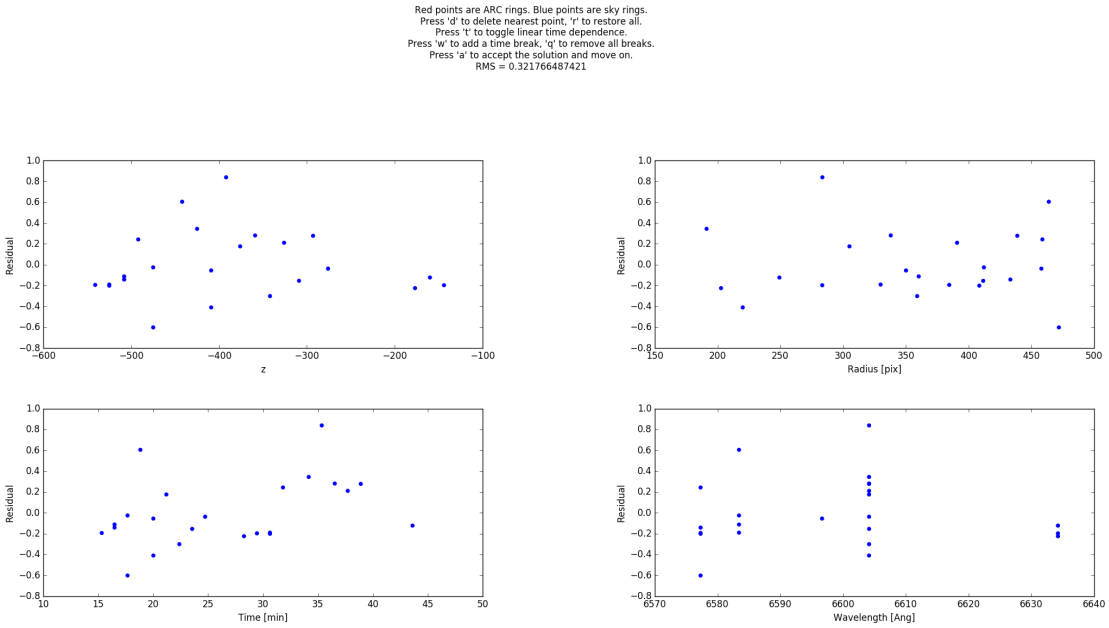
The first thing I notice on the plot is that there's a blue point with particularly large residual. This is the point at the top right of the radius plot. I'll start by removing it (by pressing 'D' over it) and see how the fit improves.



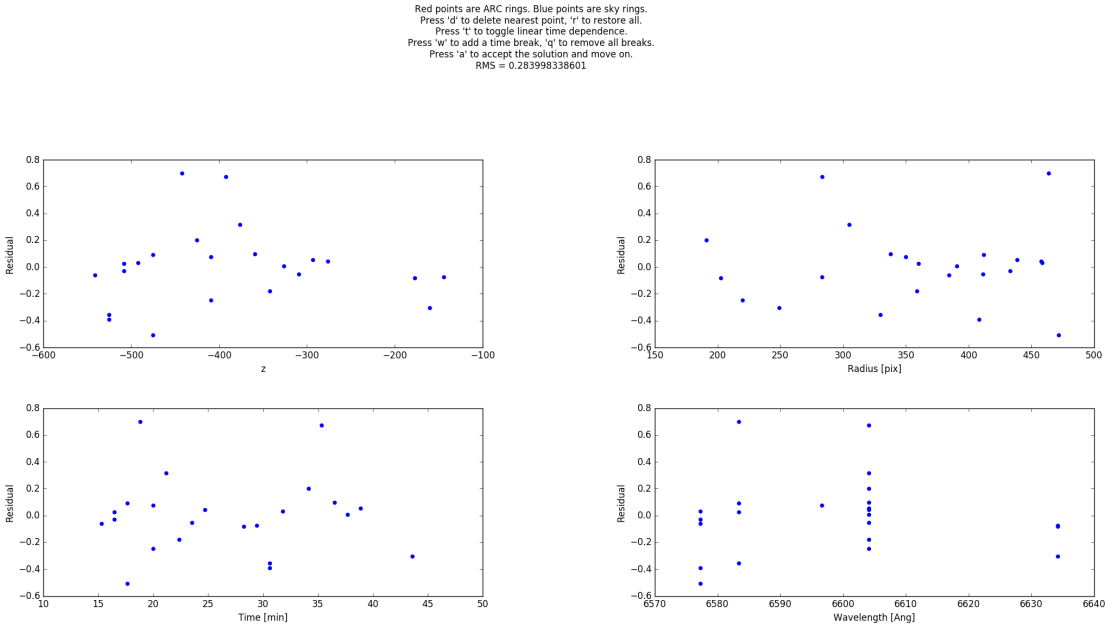
The next thing I notice is that there's a cluster of points at low-`z` with a very strong trend in their residuals vs. `z`. The ARC (red) points at high-`z` are pulling the fit, and I don't actually care about the wavelength solution at high-`z` (since I don't have any data images in that range). I remove all of these red points by pressing 'D' over each of them.



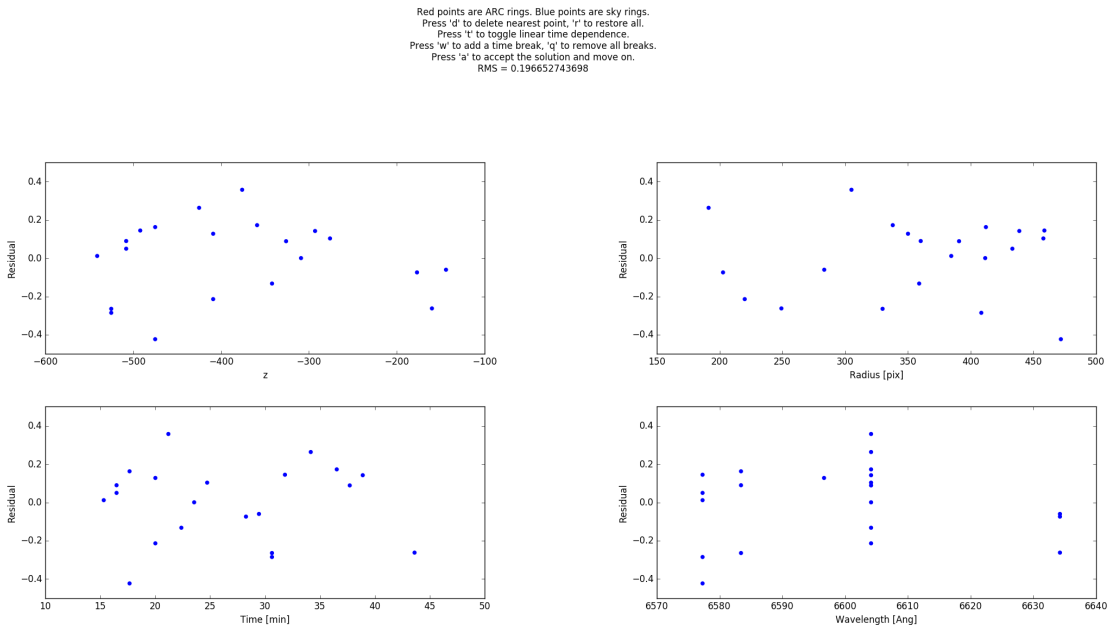
Things are improving quite a bit. I note now that the remaining two ARC (red) points are still dragging the fit in a way which is inconsistent with the blue (data) points. Since what I really care about is the wavelength solution in the data images, I decide to try removing these last two ARC points. If I later decide that was a mistake, I can restore all of the removed points by pressing 'R'.



Things are looking even better now that the ARC points are all gone. Now I notice there's a slight trend in residual vs. time, so I decide to turn on a time-dependent fit by pressing 'T'.



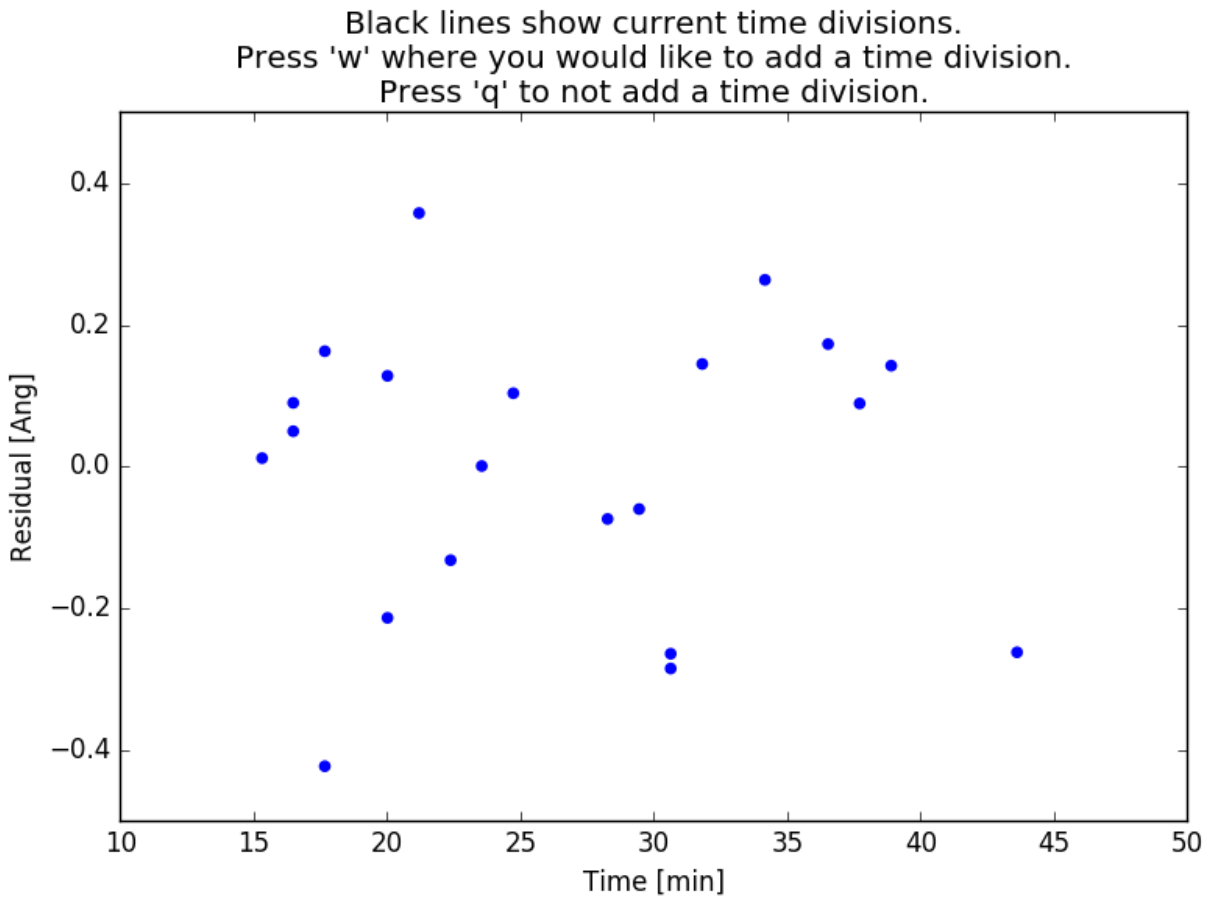
There's now two points (the ones with largest positive residual) which are pretty clear outliers from the rest of them, so I decide to remove them as well.



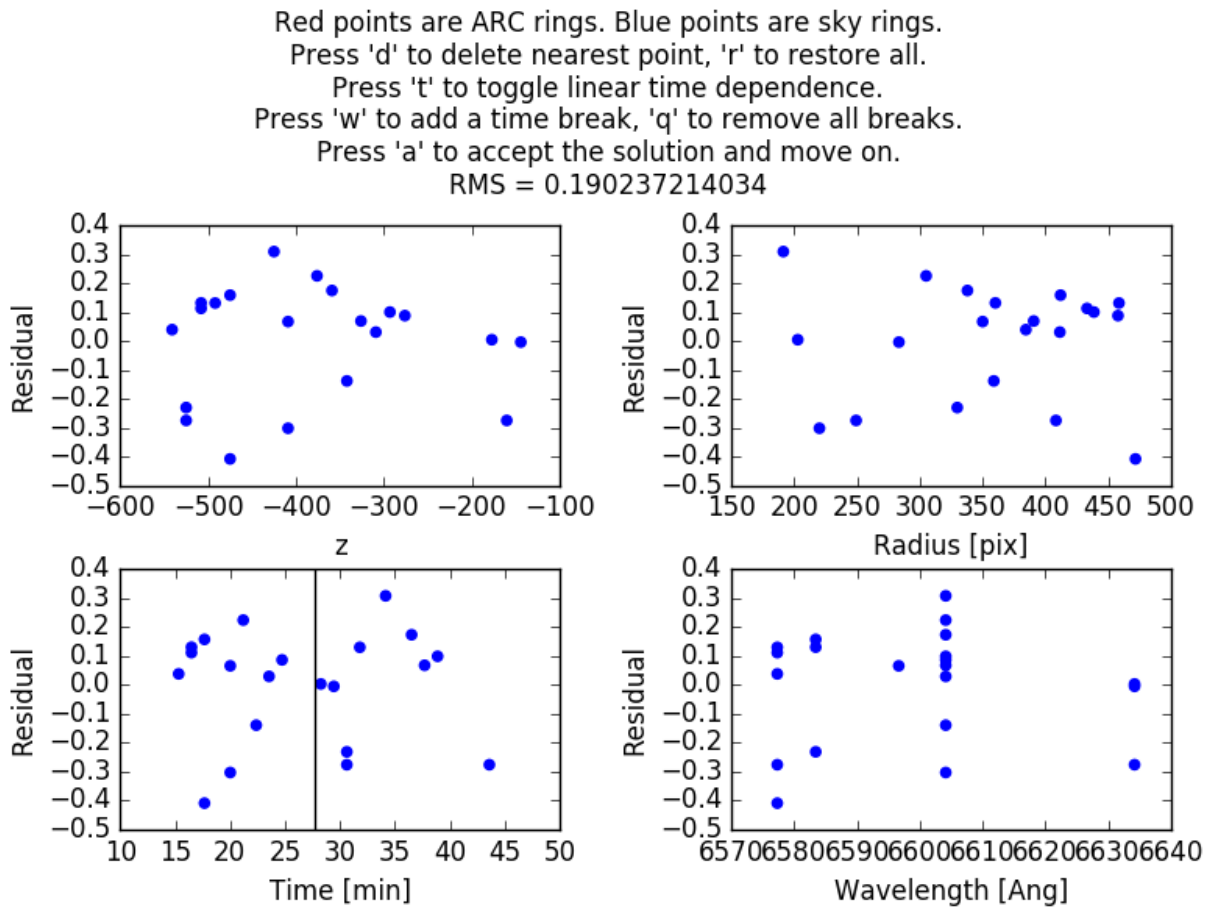
Finally, this is a wavelength solution I'm pretty happy with, so I press 'A' to accept the fit.

There's one more feature I haven't yet shown, which is the ability to add a piecewise break in time. Doing so would be unnecessary for these data, but I'll illustrate it here anyway.

If your observation sequence contained a very large jump in wavelength part-way through (for example, if you chose to interleave wavelengths), it's possible that the wavelength solution jumped a bit due to some non-linear effects. If you notice that there's a sharp break in your time residual plot, you can add a piecewise break in time by pressing 'W'.



This creates a large version of the time residual plot. Place your mouse cursor where you'd like the time break to be and press 'W' again.



Note the black line which has appeared on the time residual plot. Now two wavelength solutions are being fitted: one for points to the left of that black line and another for points to the right. This feature should only be used if there's a very clear break in your wavelength solution.

After pressing 'A' to accept the fit, the parameters of the solution will be output to the terminal. For example, I receive the following:

```
Solution 1: A = 6660.29266739, B = 0.125188184314, E = 0.0192245434506, F = 5582.20892795
Residual rms=0.196652743698 for 1 independent 4-parameter fits to 21 rings.
Accept wavelength solution? (y/n)
```

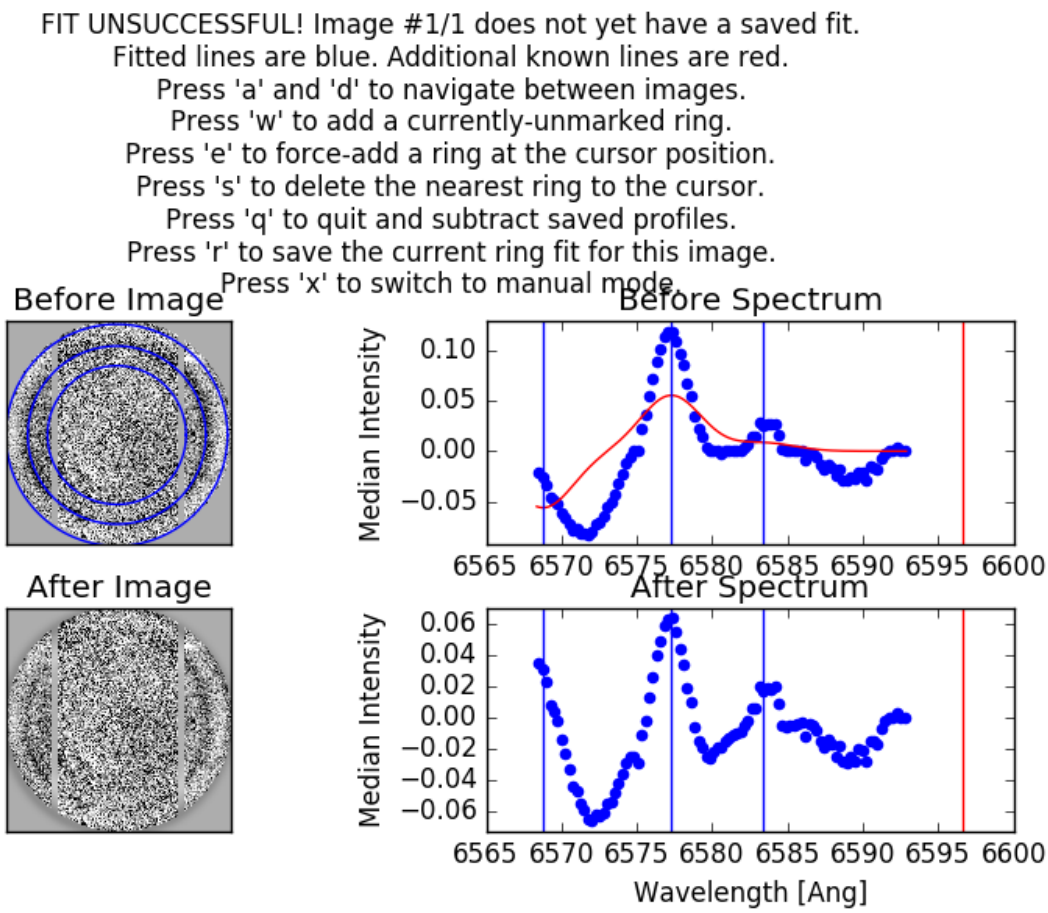
The wavelength solution for each image is recorded in its image headers as the keywords 'FPWAVE0' and 'FPCALF'. The 'FPWAVE0' keyword stores the combined value `A + B*z + E*t` and 'FPCALF' stores the value of `F`.

If you're having trouble fitting a correct wavelength solution, you can manually set the solution by editing these header keywords.

The values for the residual RMS, number of rings, number of parameters, and number of independent fits are also stored in the image header keywords 'FPCALRMS', 'NCALRING', 'NCALPARS', and 'NCALFITS'.

1.2.13 Sky Ring Subtraction

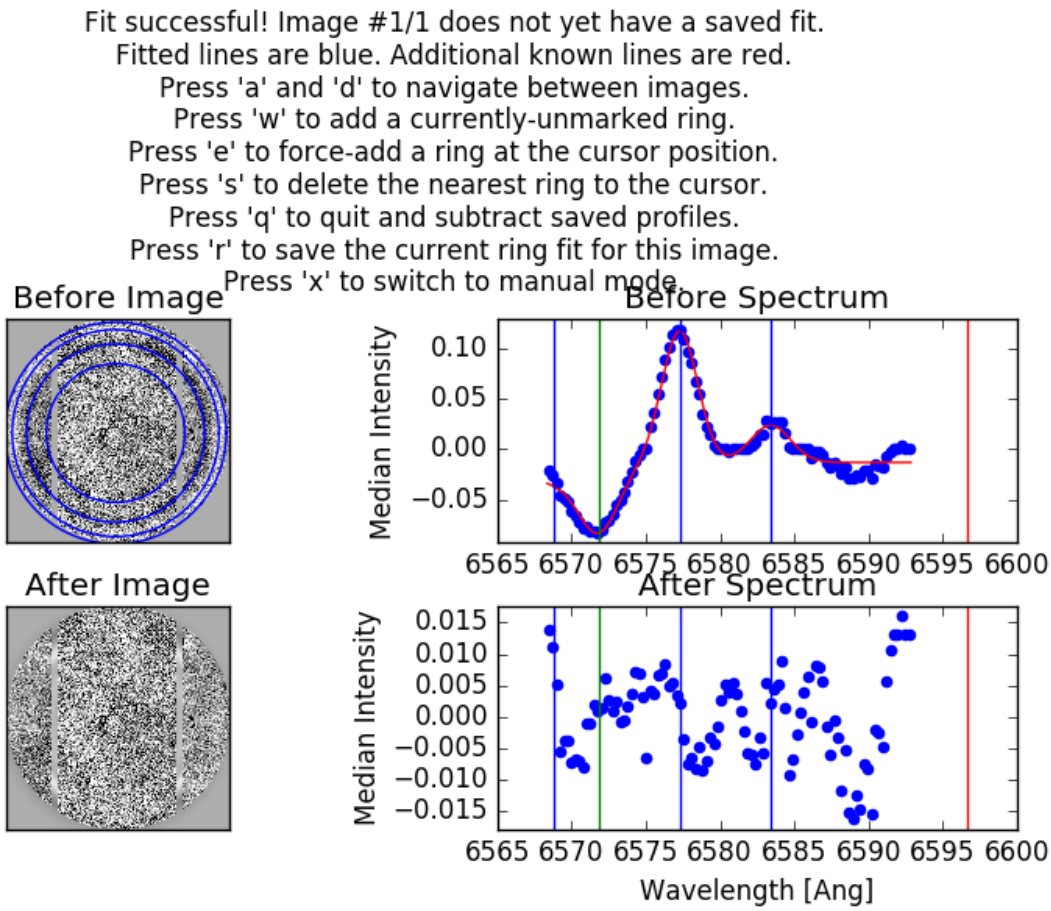
Now that you've used those pesky sky rings to fit for a wavelength solution, they've got to go! This is probably the most complicated part of the pipeline and there's some more interactive plots involved.



The top two panels you'll see are similar to the plots you saw in the wavelength calibration section – plots of median-subtracted images and azimuthally averaged radial intensity profiles. Now, however, the radial intensity profiles have been converted from radius space to wavelength space using your wavelength solution. Known night sky emission lines are denoted by vertical red or blue lines on this plot.

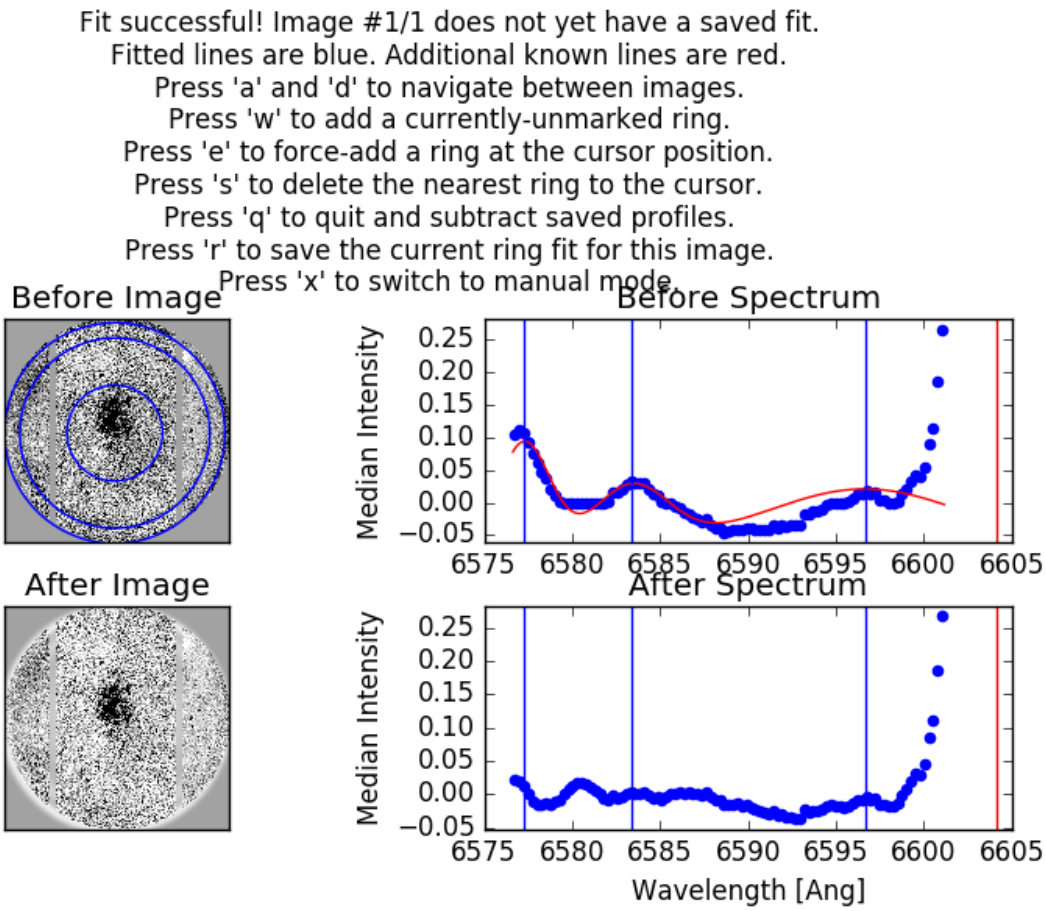
The pipeline attempts to fit a sum of Gaussian functions to this profile, with the Gaussians centered on each of the known emission lines. This fitted attempt is plotted as a red curve. The bottom two plots show the median-subtracted image after subtracting the fitted profile and its radial intensity profile.

In the above figure, you can see that the marked wavelengths are in roughly the correct positions, but the fit has failed to converge. I decide to add a "fake" wavelength by pressing 'E' at about 6572 Angstroms.

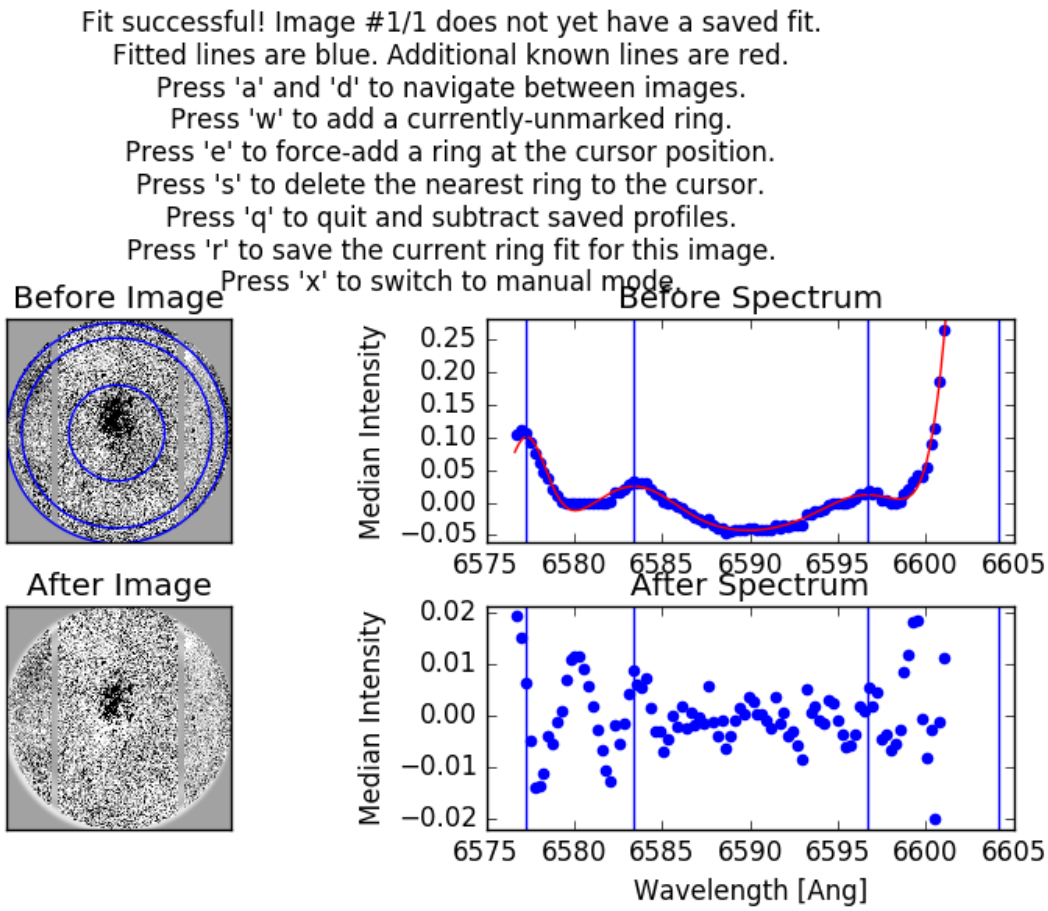


You can see that while there's now an unphysical line included in the fit (with a negative intensity!), the fit does a fantastic job. In my experience, these types of unphysical lines are often necessary to force a fit to converge. I am happy with this fit, so I press 'R' to save the fit and 'D' to move on to the next image.

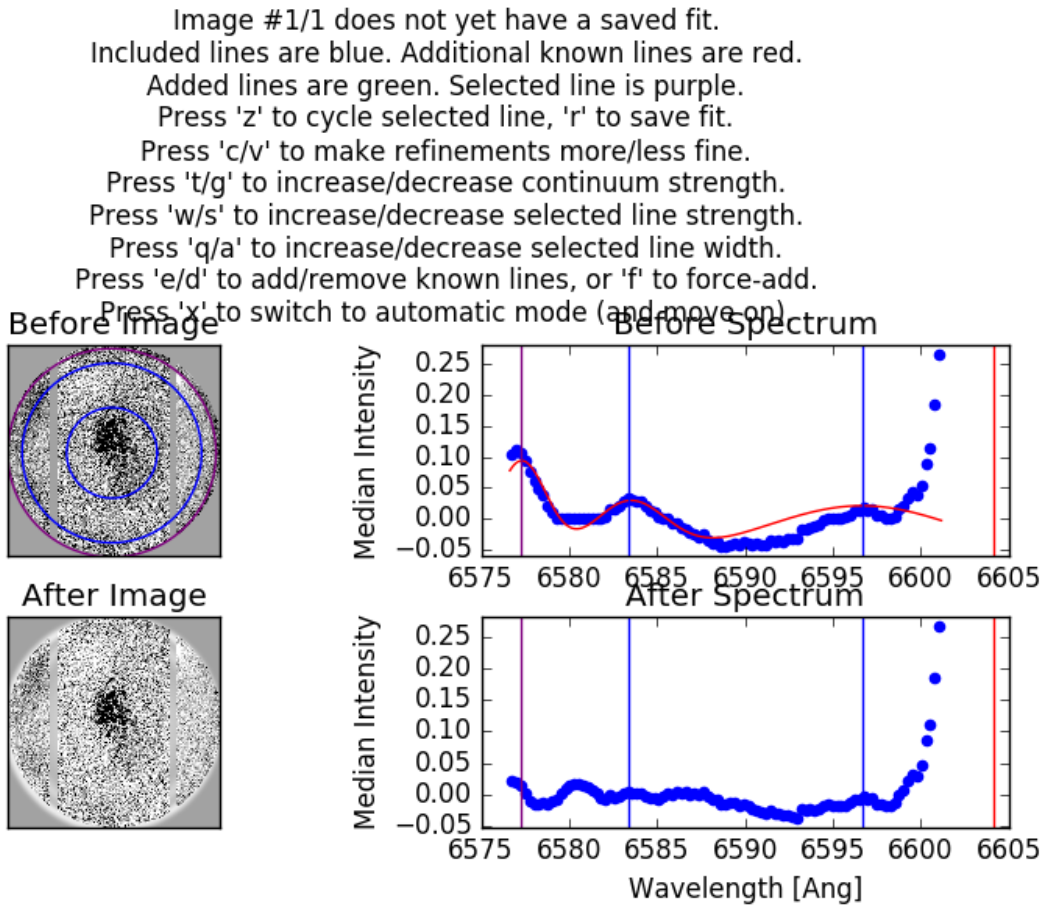
When you press 'D' to move on, the fitted rings are subtracted from the image and the image is saved and closed.



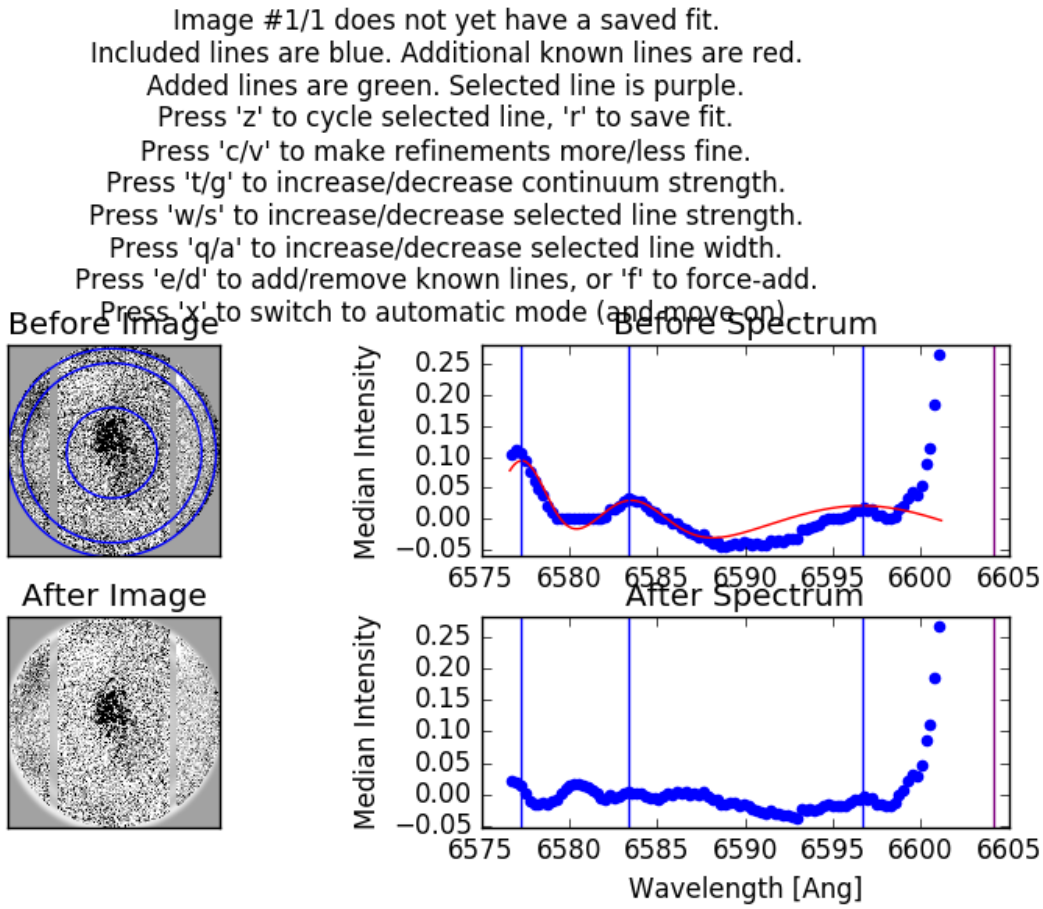
In this image, a sky ring is outside the range of the image's wavelengths, but is likely still bleeding emission into the center of the image. I can press 'W' over the vertical red line to add it to the fit.



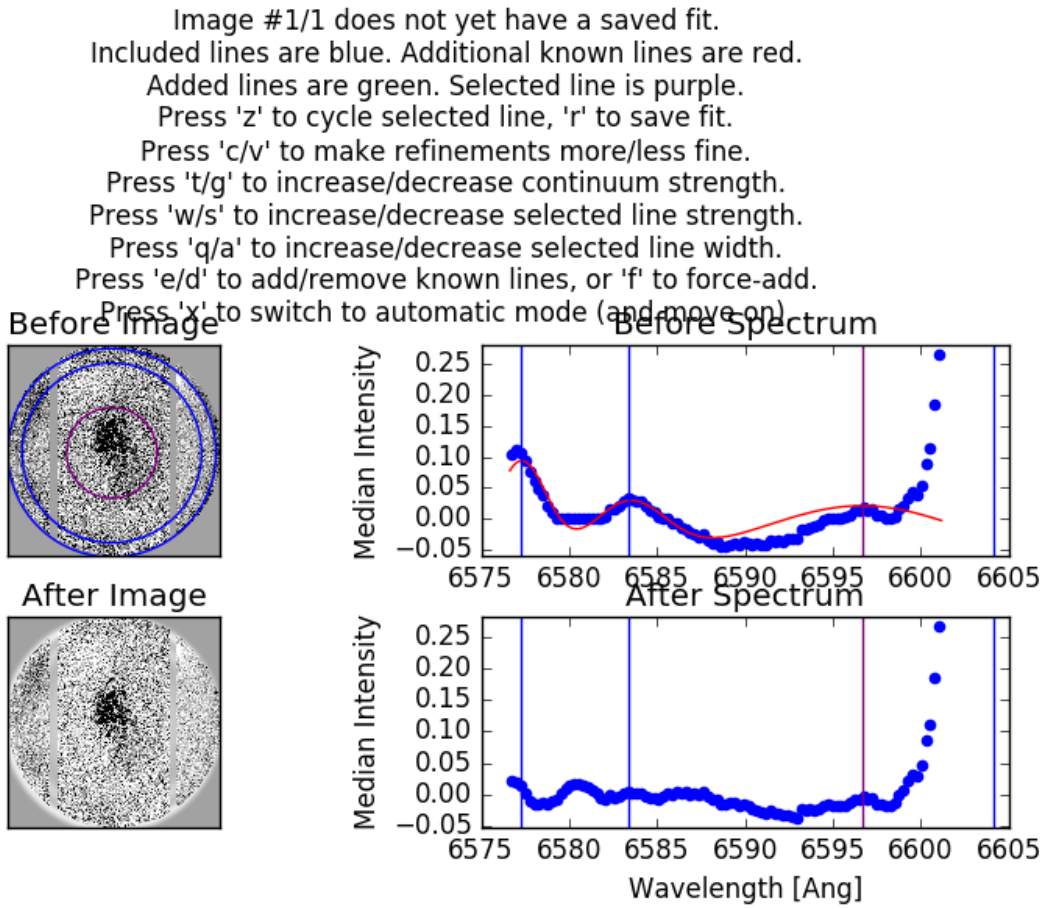
The line has been added, but I'm still unsatisfied with the fit. The sharp increase in intensity at the center of the image is likely due more to the galaxy than the night sky line. I don't want to subtract the galaxy's emission. For scenarios like this (which are hopefully rare), there is a manual ring fitting mode available. I remove the ring I added by pressing 'S' and then I press 'X' to enter the manual mode.



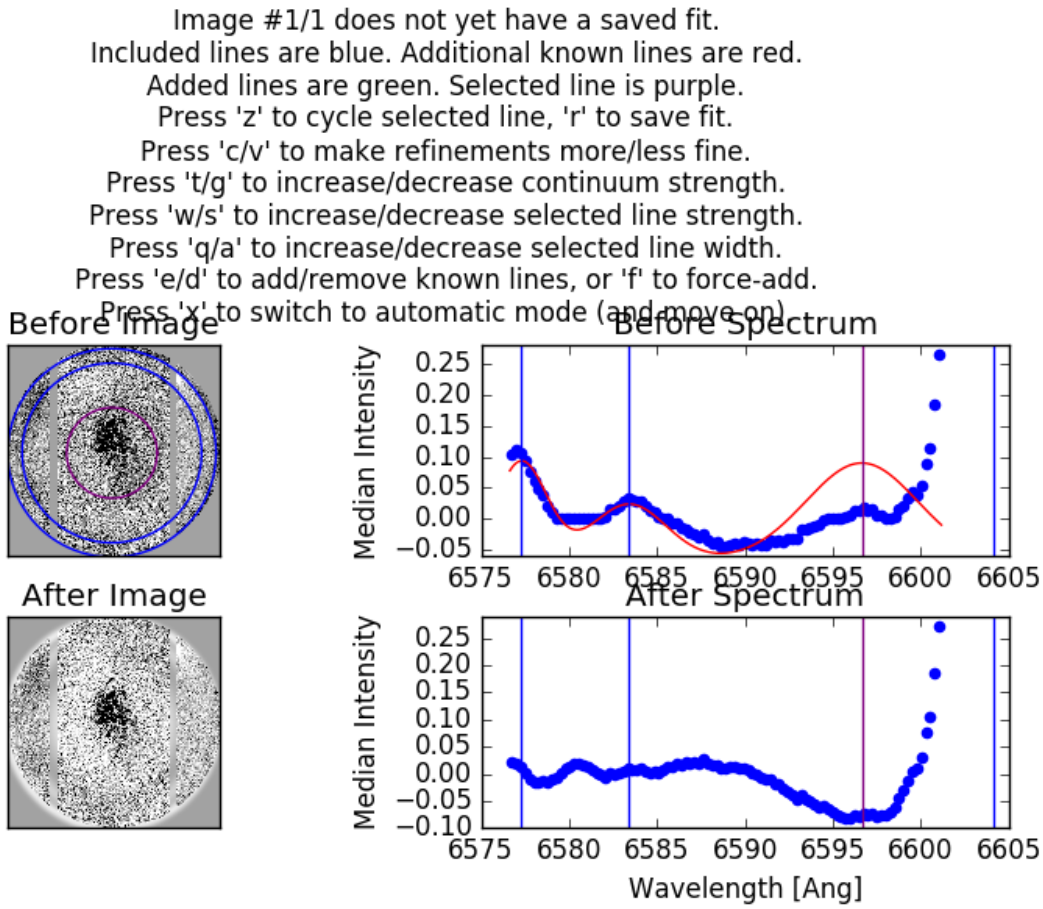
In manual mode, you can adjust the height and width of a single line. One of the lines is selected (purple) and the selected line can be cycled by pressing 'Z'. I press 'E' to add the known 6604 line back to the fit, and the selected line automatically shifts to this newly added line.



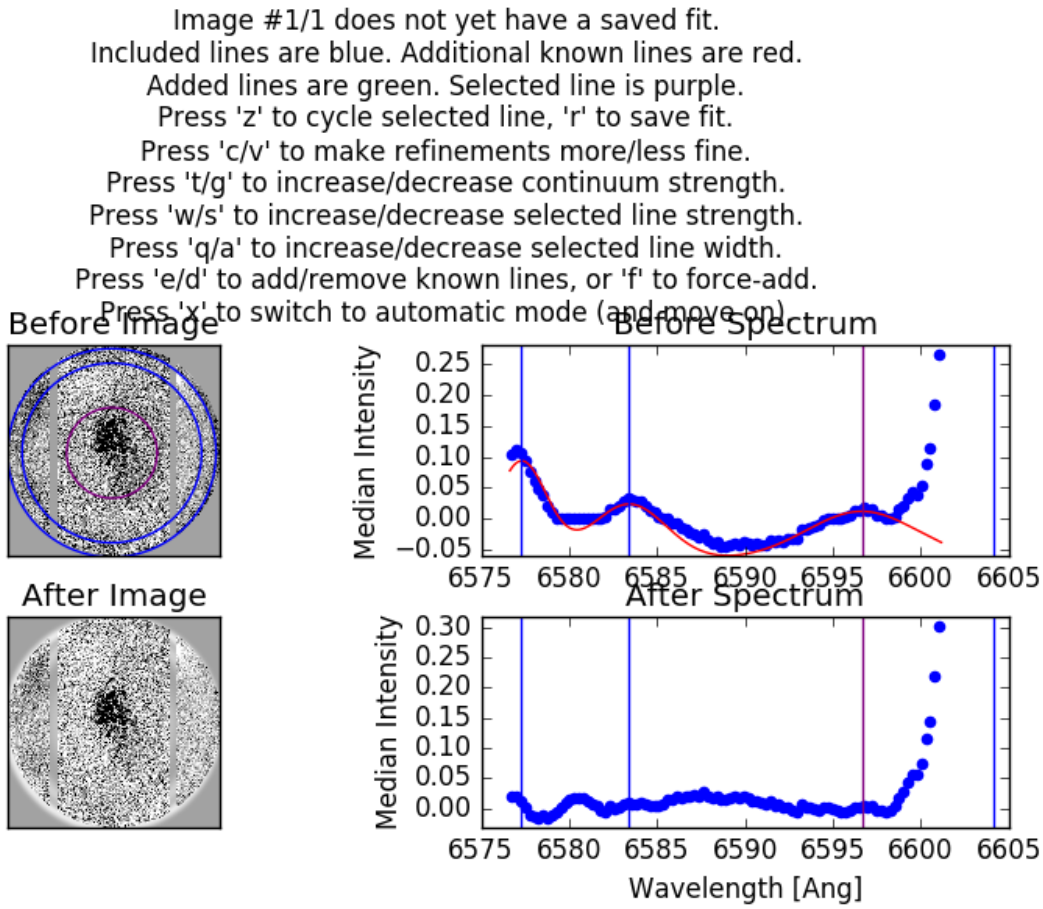
I want to adjust the line at 6596 Angstroms first, as it's currently too wide, so I press 'Z' to cycle to that line.



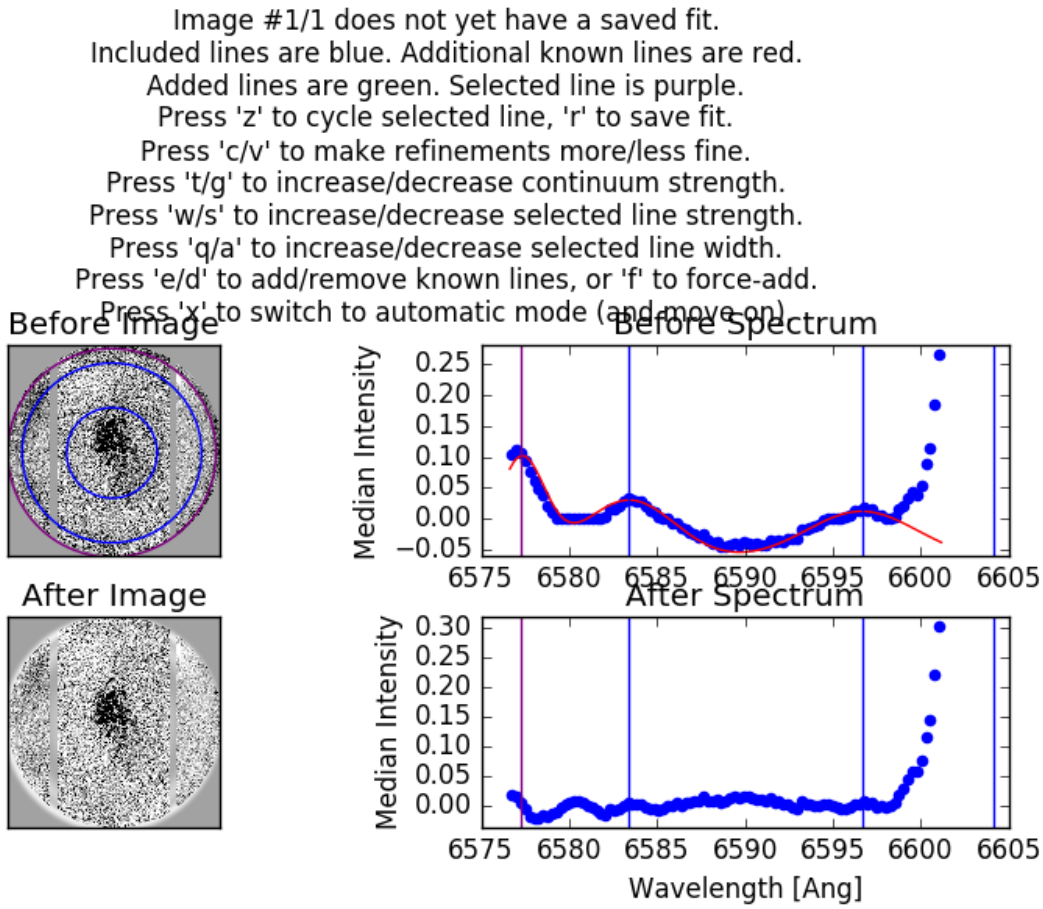
The ‘Q’ and ‘A’ keys are used to increase or decrease the selected line’s width. I press ‘A’ a few times to decrease the width of the line.



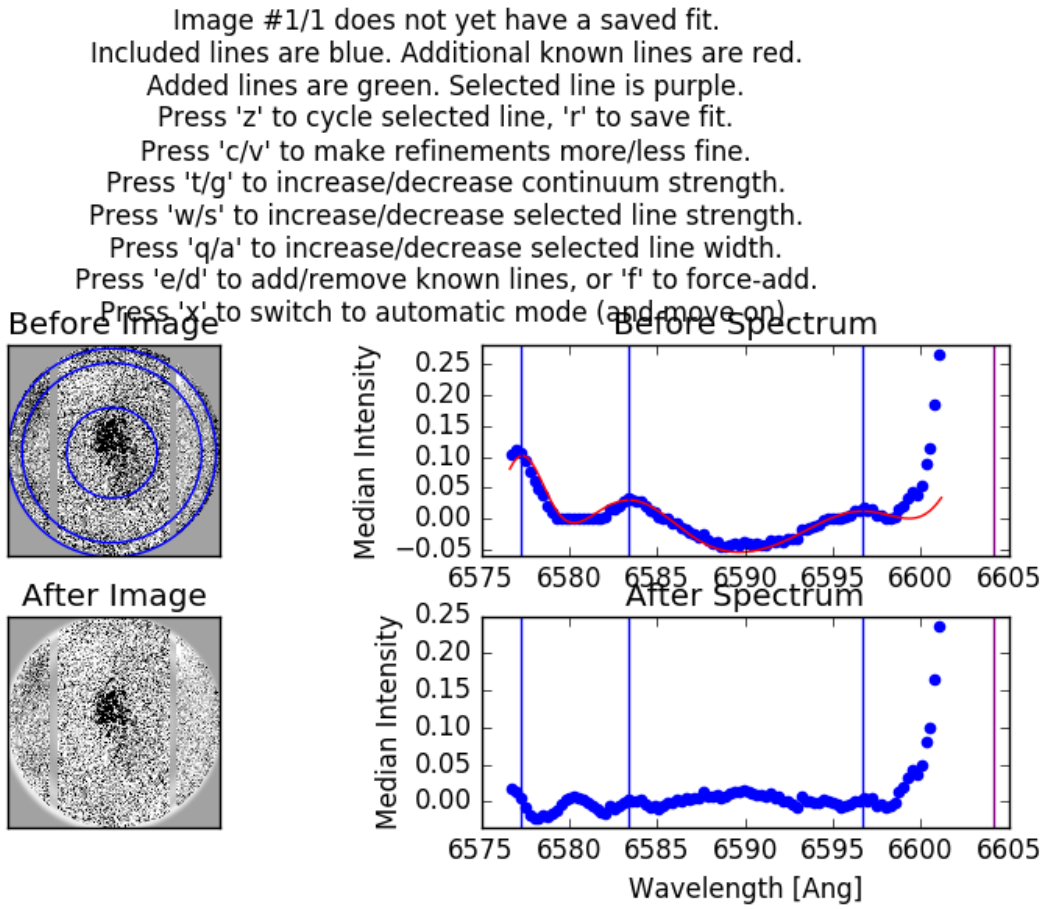
Because the *integral* of the line is held fixed, decreasing the line's width increases the height of the line. The 'W' and 'S' keys are used to increase or decrease the selected line's strength. I press 'S' a few times to decrease the strength of the line.



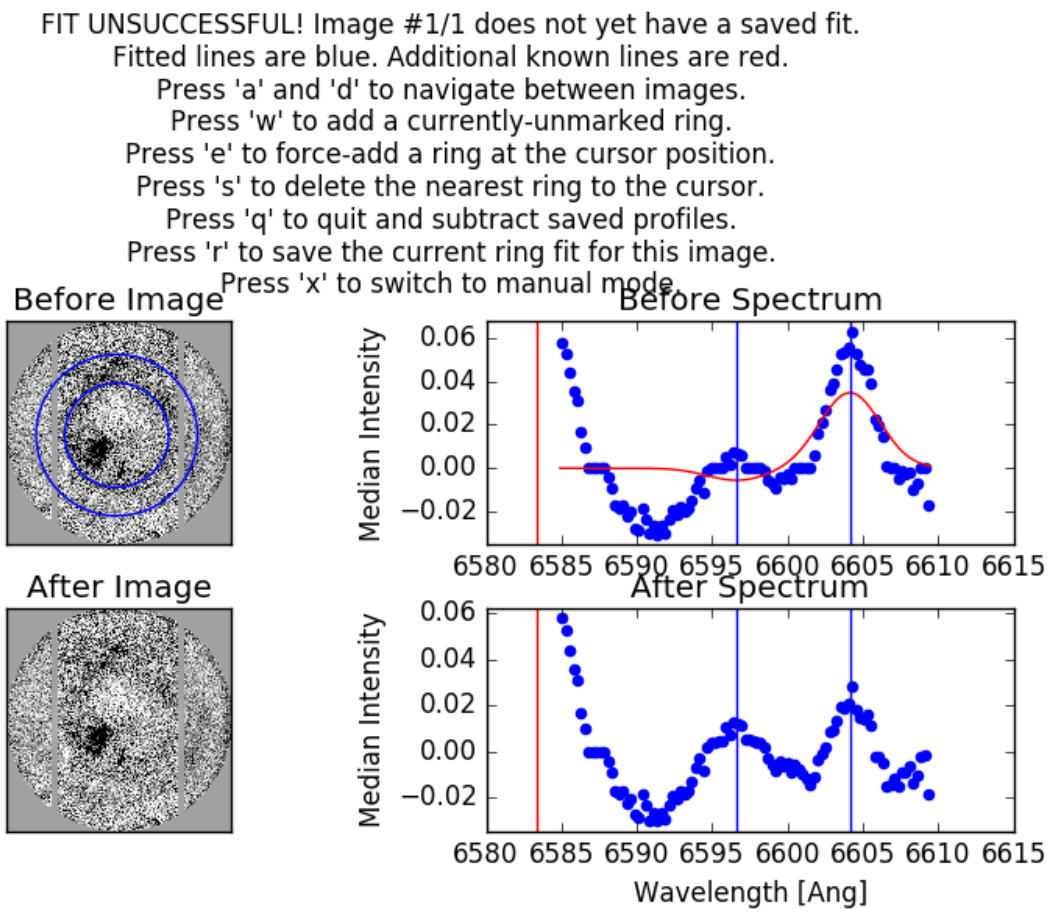
That's better! I repeat this process on some of the other lines to improve the overall fit.



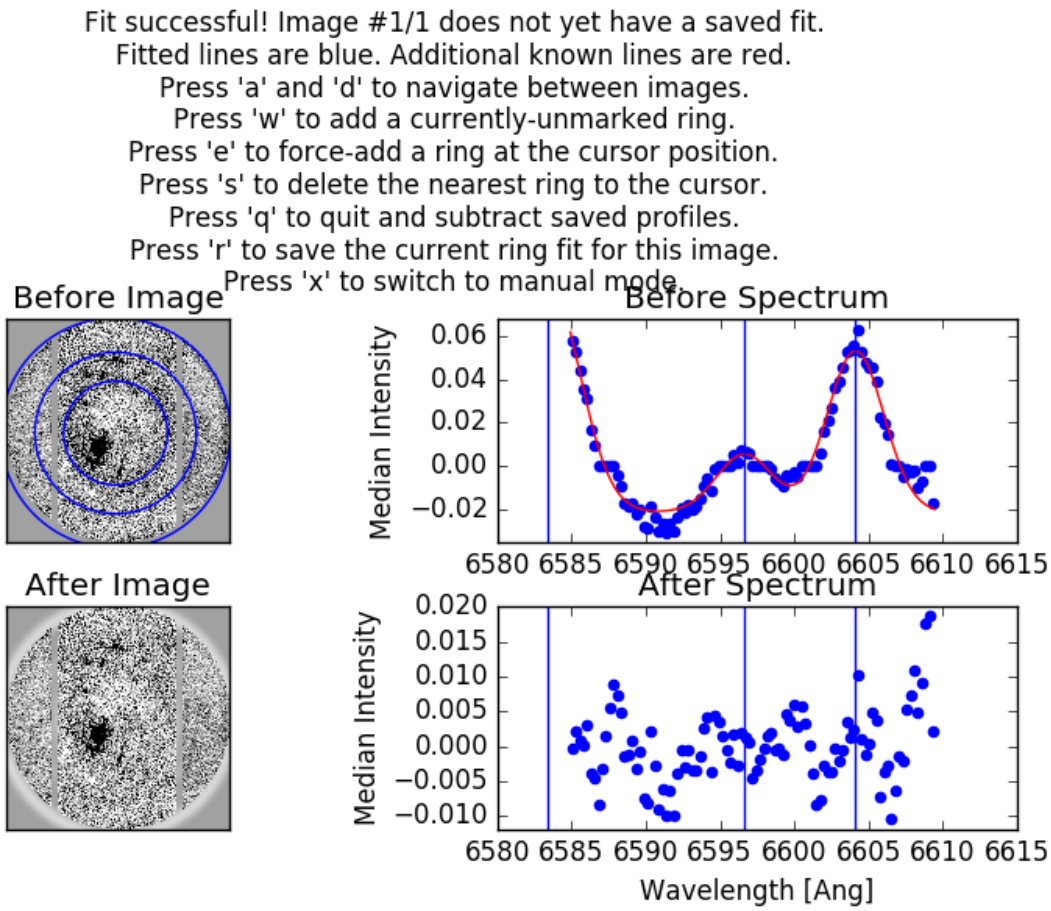
Much better. I now cycle to the newly added line to adjust its fit. If at any time the adjustments I'm making are too large or too small, I can press 'C' or 'Z' to scale the size of the adjustments by a factor of 2.



I've now adjusted the newly added line to a point where I think it has subtracted the night sky ring emission without subtracting from the galaxy. I'm pretty content with this fit, so I press 'R' to save it, then 'D' to move on to the next image. Note that when I press 'R', it returns me to the automatic plotting view rather than the manual mode. While the manual fit is no longer plotted, its parameters have been saved and it's save to move on.



Here's another example where there's a real night sky line that's outside our image's wavelength range but is likely close enough to bleed into it. As before, I add it to the fit by pressing 'W' over the vertical red line.



Unlike last time, this time I got a good fit by added this line. Time to save it (by pressing 'R') and move on (by pressing 'D').

All that's left to do now is repeat this process for each image in your observation sequence. This can be a time-consuming task, but cleanly removing this night sky emission is an important task. With a bit of experience, this stage of the data reduction process gets a *lot* easier.

Note: The parameters of the fitted and subtracted rings are saved in the image headers should you ever need them again. The header keywords 'SUBWAVE#', 'SUBINTY#', and 'SUBSIG#' record the wavelength, intensity, and Gaussian width of each subtracted line, where '#' is an index referring to the subtracted line (0, 1, 2, etc.).

1.2.14 Data Cube Creation

Once sky rings have been subtracted, it's time to create the final data cube!

The pipeline will prompt you for a desired seeing FWHM in the terminal. The default value displayed will be equal to the worst seeing of all of your FP images. If you desire a larger FWHM (e.g. to increase signal to noise by convolving with a larger beam), you can enter it here.

The code will then create a new directory for your data cube, called '(YourObject)_cube' and begin generating the data cube images there. For each image, an appropriate Gaussian convolution kernel will be created which will blur your images to the desired seeing and shift them using the shifts calculated earlier. These blurred images (and their uncertainties and wavelengths) will be written to the data cube directory.

For more information about the images this step creates, see the section about the FPImage class.

1.2.15 Solar Velocity Shifting

This routine calls the IRAF task 'rvcorrect' to calculate the velocity of SALT towards or away from your object relative to the sun, then corrects the wavelengths in the data cube to adjust for the Doppler shift.

This should require no user intervention.

1.2.16 Velocity Map Fitting

This step is only called if you have compiled the optional voigtfit module and should require no user intervention. The pipeline attempts to fit a double Voigt profile at each pixel of the data cube (currently this is hard-coded to be the H-alpha and [NII] 6583 lines). The parameters of the fit are saved as a set of FITS images in the data cube directory:

- continuum.fits – The fitted continuum strength
- intensity.fits – The fitted H-alpha intensity
- wave.fits – The fitted central wavelength of the H-alpha line
- sigma.fits – The fitted Gaussian width of the profile
- n2intensity.fits – The fitted [NII] line intensity
- chi2.fits – The reduced-chi-squared statistic of the fit
- dcontinuum.fits – The uncertainty in the fitted continuum strength
- dintensity.fits – The uncertainty in the fitted H-alpha intensity
- dsigma.fits – The uncertainty in the fitted Gaussian width sigma
- dn2intensity.fits – The uncertainty in the fitted [NII] intensity
- n2ratio.fits – The ratio of the [NII] intensity to the H-alpha intensity
- velocity.fits – The line-of-sight velocity at the pixel, assuming the wavelengths correspond to H-alpha emission
- dvelocity.fits – The uncertainty in the fitted velocity

1.2.17 Velocity Map Masking

Finally, this step allows for a series of cutoffs to be made in the fitted velocity map to mask undesirable pixels. The currently supported cutoffs are:

- Minimum Signal-to-Noise Ratio (dIntensity / Continuum)
- Minimum Velocity
- Maximum Velocity
- Maximum Chi^2
- Maximum Relative Uncertainty in Sigma (dSigma / Sigma)
- Maximum Relative Uncertainty in Intensity (dIntensity / Intensity)

1.3 Image File Structure

1.3.1 FITS Images

The image files created by this code are multi-extension FITS images, which have a few headers and data arrays in each file. If you are interacting with these files using IRAF or the Astropy/Pyfits packages, you'll need to know which extensions are which.

- Extension 0: This extension has no data array. It's header is the primary header of the file and is where all of the important keywords are stored.
- Extension 1: This extension contains the primary data array, i.e. your image.
- Extension 2: This extension contains the variance array. Each pixel in this array corresponds to the variance of a pixel in your image.
- Extension 3: This extension contains the bad pixel mask array. Each pixel is either 0 or 1. Pixels with values of 1 are bad.
- Extension 4: If this extension exists, it contains a wavelength array. Each pixel corresponds to the wavelength of a pixel in your image array.

Here's a list of some important header keywords created by *saltfppipe* that you might want to know about:

- FPAXCEN / FPAYCEN / FPARAD – The center and radius of the circular aperture mask.
- FPFWHM – The average full-width half-max of stars in the image.
- FPXCEN / FPYCEN – The location of the ghost reflection center / optical axis of the etalon.
- FPWAVE0 / FPCALF – The fitted wavelength solution parameters.

1.3.2 The FPImage Class

This software also includes a class, `FPImage`, for interacting with the images created by *saltfppipe*. The class is largely a wrapper around a number of astropy / numpy methods.

To load an image file:

```
>>> from saltfppipe.fp_image_class import FPImage
>>> im = FPImage('mfxgbpP201111010208.fits', update=True)
```

This loads the image file `mfxgbpP201111010208.fits` into the variable `im`. The `update=True` argument will make it so that when I close the image, any changes I've made will be saved. The data arrays are located in the class attributes `inty`, `vari`, `badp`, and `wave`. These are the pixel intensity, variance, bad pixel mask, and wavelengths, respectively.

Note: The wavelength array extension is created at a relatively late stage of the pipeline. If it does not exist, `im.wave` will be `None`.

For an example of how you can interact with these arrays, let's say I wanted to set the intensity values in my image to 0 wherever there are bad pixels:

```
>>> im.inty[im.badp == 1] = 0
```

The `im.writeto(filename, clobber=False)` method will write the image to a file specified by filename. Unless `clobber` is set to True, it will not overwrite an existing file.

There are some useful class methods for getting arrays which are the same *shape* as your image and contain pixel coordinates.

```
>>> xarray, yarray = im.xyarrays()
>>> rarray = im.rarray(798, 510)
```

The first of these methods returns two arrays. `xarray` contains the X coordinate of every pixel in the image. `yarray` does the same for the Y coordinates.

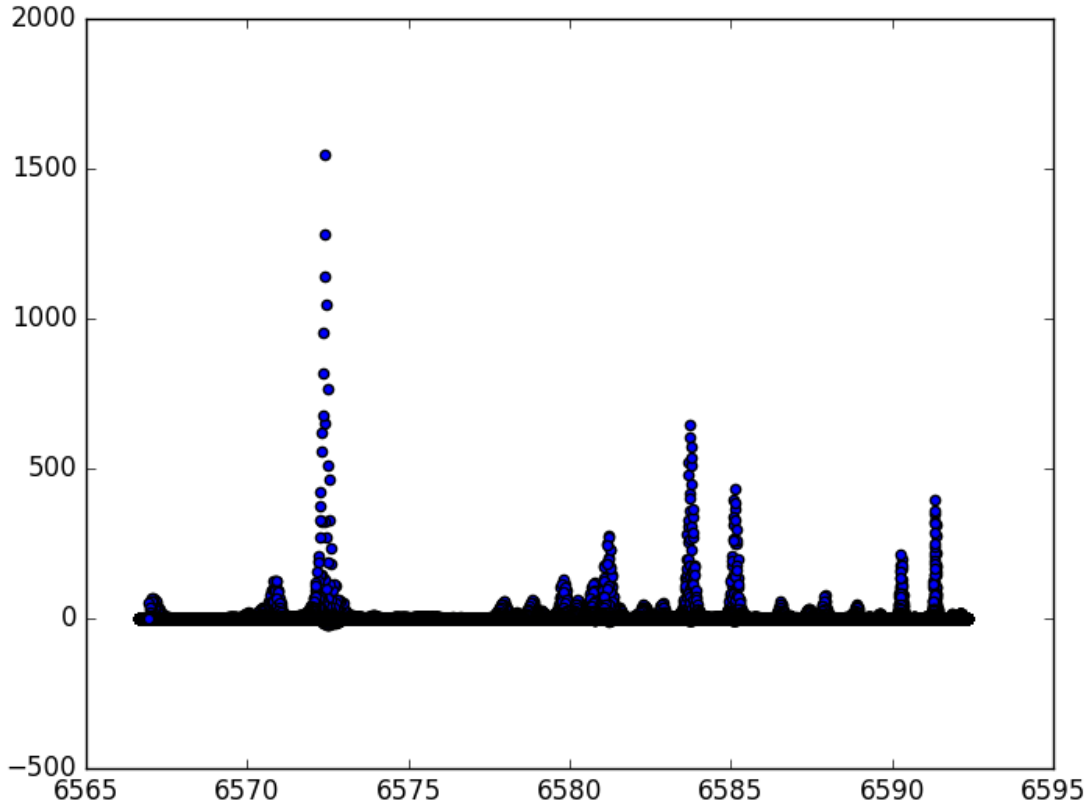
The second method, `rarray`, returns the *radius* of every pixel in the image from the point which you specify in the calling arguments. In this case, `rarray` contains the radius of every pixel from the point (798, 510).

You can get at several of the important header keywords using class attributes:

- `im.axcen` – The X coordinate of the aperture center
- `im.arad` – The radius of the circular aperture
- `im.ycen` – The Y coordinate of the ghost reflection center
- `im.object` – The name of the object which was observed
- `im.fwhm` – The average FWHM of the stars in the image

As an example of how to put all of this together, let's create a plot of pixel intensity vs. wavelength for the pixels in my opened image. Furthermore, let's exclude any pixels which are outside the circular aperture as well as any pixels with values of '1' in the bad pixel mask:

```
>>> # Some useful imports
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> # Get the radius from the aperture center at every pixel
>>> rarray = im.rarray(im.axcen, im.aycen)

>>> # Create a boolean array which is True for pixels
>>> # inside the aperture and False outside it
>>> goodpix = rarray < im.arad

>>> # Combine this boolean array with the bad pixel mask
>>> goodpix = np.logical_and(goodpix, im.badp != 1)

>>> # Plot the intensity and wavelength of pixels wherever
>>> # 'goodpix' is True
>>> plt.scatter(im.wave[goodpix], im.inty[goodpix])
>>> plt.show()
```



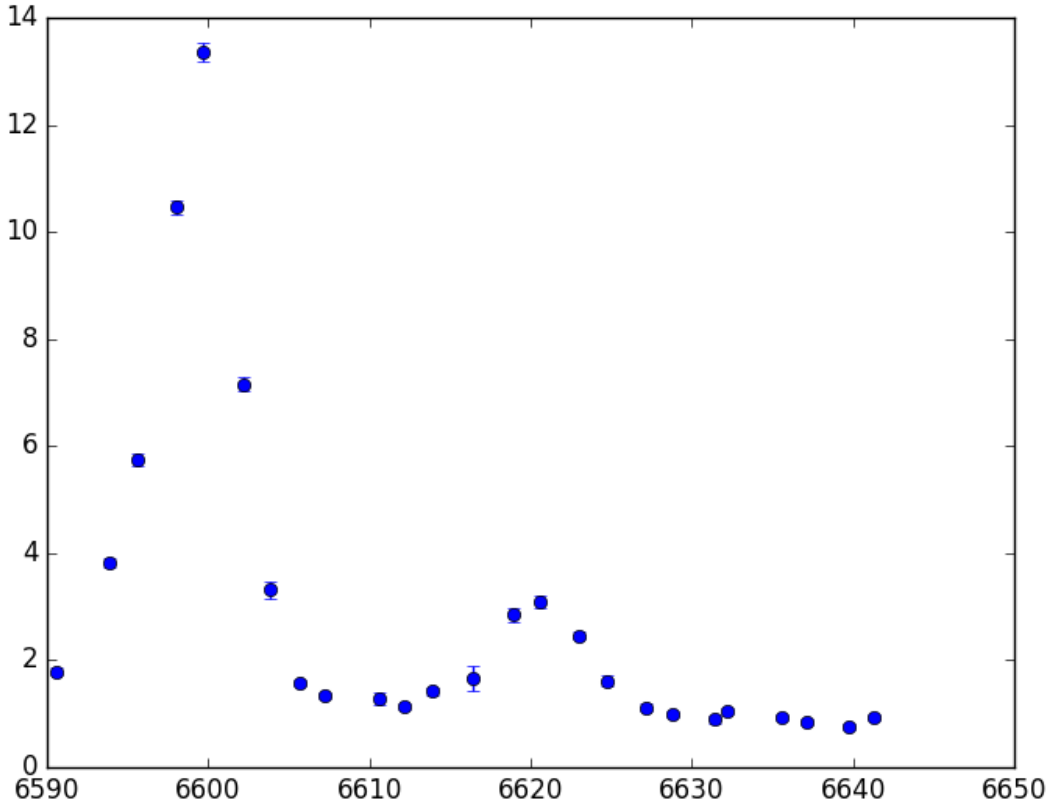
In this plot we can see a number of local peaks, which likely correspond either to galactic H-alpha and [NII] emission or to night sky emission lines.

By calling the image's `im.close()` method, we can close the image and save any changes we made to it.

As another example, let's say I want to look at the spectrum of my data cube at a the pixel ():

```
>>> # Some useful imports
>>> from saltfppipe.fp_image_class import FPImage
>>> import matplotlib.pyplot as plt
>>> from os import listdir
>>> import numpy as np

>>> # Get a list of files in the data cube directory
>>> cd NGC2280_cube
>>> fnlist = sorted(listdir('.'))

>>> # Open all of these files as FPImage objects
>>> images = [FPImage(fn) for fn in fnlist]

>>> # Choose the pixel we want to look at
>>> x, y = 780, 627

>>> # Get the spectrum at that point (if the pixels aren't bad!)
>>> badps = np.array([im.badp[y, x] for im in images])
>>> intys = np.array([im.inty[y, x] for im in images])[badps != 1]
>>> varis = np.array([im.vari[y, x] for im in images])[badps != 1]
```

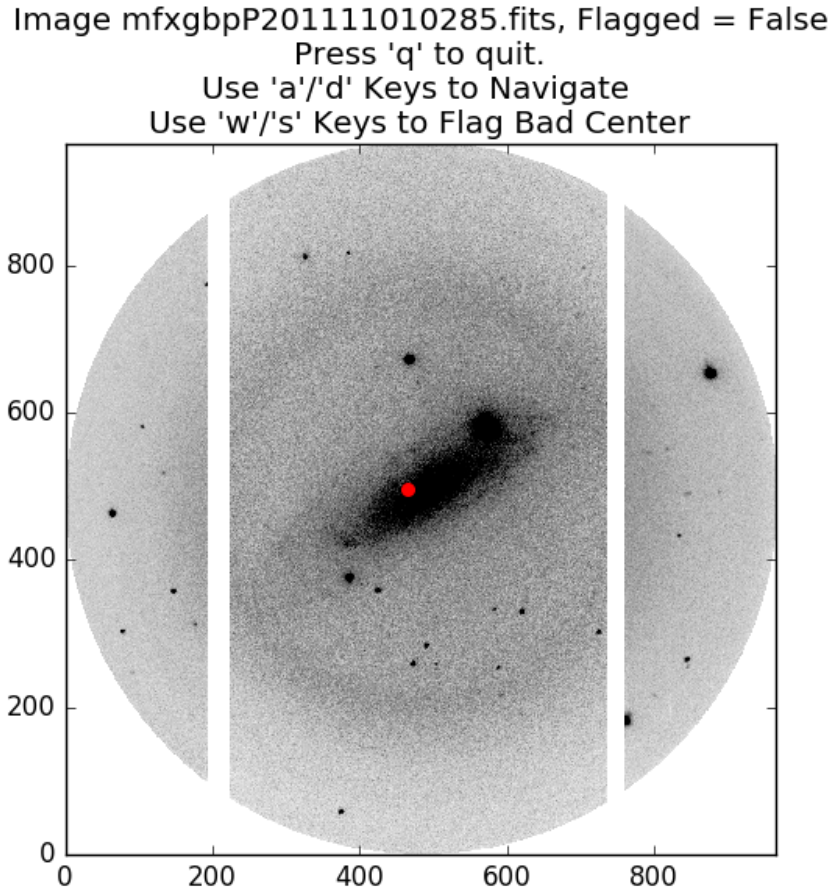
```
>>> waves = np.array([im.wave[y, x] for im in images])[badps != 1]

>>> # Plot the spectrum
>>> plt.errorbar(waves, intys, yerr=np.sqrt(varis), fmt='o')
>>> plt.show()

>>> # Closing images is good practice!
>>> for im in images: im.close()
```



1.4 Useful Functions

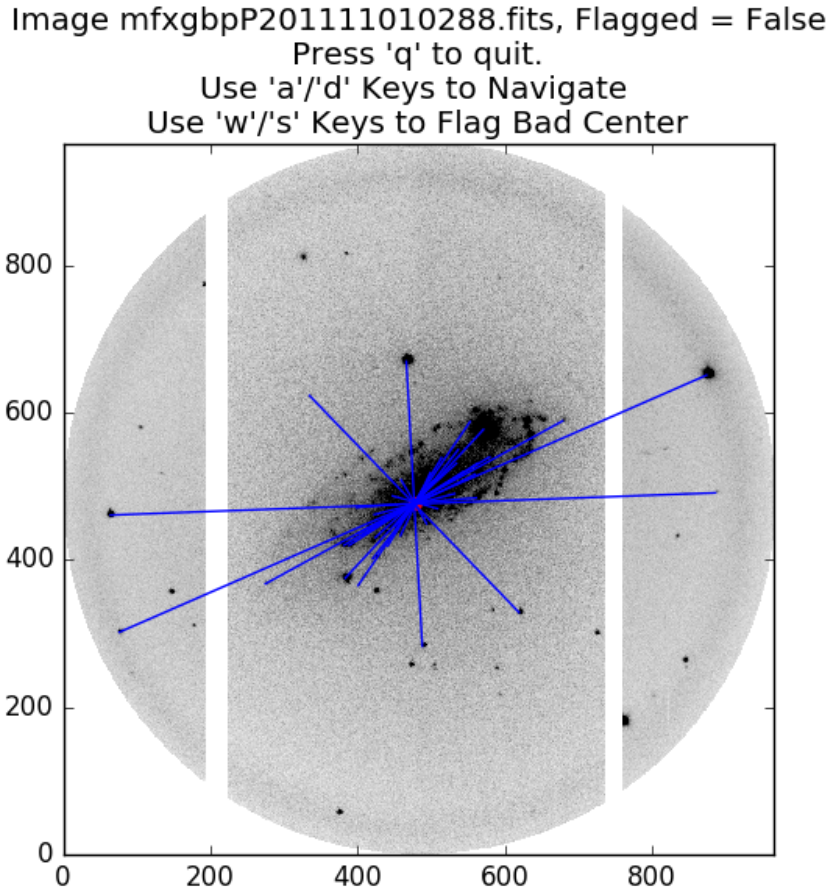
1.4.1 find_ghost_centers

In sparse star fields, the default ghost center finding routine occasionally fails. When this happens, you can use the function 'find_ghost_centers' to try to find the center using different input parameters.



In this image of NGC 1325, the default routine has failed to find the reflection center for the ghosts. After manually examining some of my images in SAOImage DS9, I'm pretty sure the center is somewhere around pixel coordinate [793, 503]. We can force it to find this center using the 'find_ghost_centers' routine. This is best done interactively from the python or ipython terminal:

```
>>> # The images we want are in the product directory:
>>> cd product/

>>> # A few useful imports:
>>> from os import listdir
>>> from saltfppipe.find_ghost_centers import find_ghost_centers

>>> # I happen to know that the first 3 images and last 1 image
>>> # of my 'product/' directory are ARC images and not images
>>> # of NGC 1325, so I don't include them in 'fnlist':
>>> fnlist = sorted(listdir('.'))[3:-1]

>>> # Now we call the 'find_ghost_centers' function:
>>> find_ghost_centers(fnlist, tolerance=7,
>>>                    thresh=3, guess=[792, 502])
```



Here we see that the 'find_ghost_centers' function successfully found several star/ghost pairs and located the center.

The calling keywords are:

- `fnlist` – A list of strings. Each string should be the path to an image file in which you want to detect the ghost center.
- `tolerance` – How distant two pixels can be from each other to be considered "close enough", in units of image pixels. Higher values mean the code is more likely to find a center. Extremely high values will likely match objects which aren't truly star/ghost pairs. Default value is 3.
- `thresh` – The minimum brightness required for an object to be detected, in units of the standard deviation of the sky background. Lower values will detect more objects, increasing the likelihood of finding pairs. Values which are too low will detect spurious objects. Default value is 4.
- `guess` – If you think you know the center, use this keyword. Must be an array-like object of length 2 containing the X and Y coordinates of your guess. The true center must be within `tolerance` pixels of your guess for this to work. No default value.

Note: If you're locating pixels in SAOImage DS9, subtract 1 from both coordinates before using it in Python. Python indices begin at zero, while DS9 indices begin at 1.

Note: If you've got a dense star field, this function can take a while to run. Increasing the value of `thresh` or decreasing the value of `tolerance` will speed up the routine. For dense star fields, this should still result in a good

fit while taking much less time to run.

1.4.2 flatten

The 'flatten' function is what's used by the pipeline to flatten your images, but it can also be used outside of the pipeline for a bit more control. It takes two calling arguments:

- `fnlist` – A list of strings. Each string should be the relative path to an image file you want to flatten.
- `flatfile` – A string. The relative path to a flatfield image.

Let's say you want to flatten each data image by a different flatfield image. You can accomplish this by:

```
>>> # Get your lists of data images and flatfield images somehow:
>>> data_files = dummy_way_to_get_image_list('data')
>>> flat_files = dummy_way_to_get_image_list('flat')
>>> # Note that these are not real functions. I don't
>>> # know where or how your flatfield images are stored.

>>> # Import the flatten function:
>>> from saltfppipe.flatten import flatten

>>> # Call the flatten routine separately for each data
>>> # image and its corresponding flatfield:
>>> for data_file, flat_file in zip(data_files, flat_files):
>>>     flatten([data_file], flat_file)
```

Note: Take care with the square brackets `[...]` around `data_file` in this function call. The 'flatten' routine expects a list of image file names, so we must give it a list of length 1 for this to work properly.

1.4.3 align_norm

The 'align_norm' function is what the pipeline uses to align and normalize your images to one another, but it can also be used outside of the pipeline if you need a bit more control. It has a few calling arguments:

- `fnlist` – A list of strings, Each string should be the relative path to an image file in your SALT FP observation.
- `tolerance` – How distant two pixels can be from each other to be considered "close enough", in units of image pixels. Higher values mean the code is more likely to match stars to each other if the image alignment drifted a lot over the course of an observation. Default value is 5 pixels.
- `thresh` – The minimum brightness required for an object to be detected, in units of the standard deviation of the sky background. Lower values will detect more objects, increasing the likelihood of identifying dim stars in all images. Values which are too low will likely detect spurious objects. Default value is 3.5.

The way I recommend manually calling this function depends on the problems you're encountering with the pipeline's default version of this routine:

- Too many stars (some of them spurious) are being detected: I recommend increasing the value of `thresh`.
- Very few (but nonzero) stars are being detected: I recommend decreasing the value of `thresh`.
- No stars are being detected: I recommend a combination of decreasing `thresh` and increasing `tolerance`.

As an example, the default version of this routine identifies far too many stars (several of them spurious) when applied to the RINGS observations of NGC 2280. To manually call this function on those data, I'd do the following:

```
>>> # Useful imports:
>>> from os import listdir
>>> from saltfppipe.align_norm import align_norm

>>> # Change to the object's directory and get a list of the images:
>>> cd NGC2280/
>>> fnlist = sorted(listdir('.'))

>>> # Call the align_norm function with an increased thresh value:
>>> align_norm(fnlist, thresh=5)
```

This results in far fewer stars being identified by the routine and the detections which are made being more robust.

This page was last updated by Carl Mitchell on 19 July 2016.